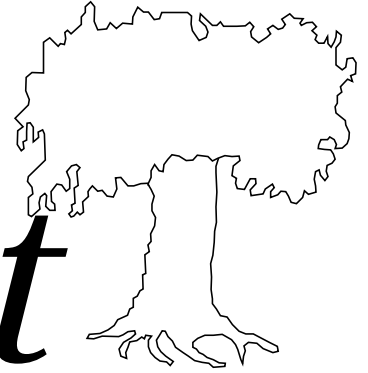


Sina/st



User's Guide and Reference Manual

Sina/st version 3.1

P.S. Koopmans

TRESE Project
Department of Computer Science
University of Twente

4 July 1996

Sina/st User's Guide and Reference Manual

P.S. Koopmans

TRESE Project
Department of Computer Science
University of Twente
The Netherlands



TRESE Project
Department of Computer Science
University of Twente
PO box 217
7500 AE Enschede
The Netherlands

Electronic mail:
sina@cs.utwente.nl

World Wide Web:
<http://wwwtrese.cs.utwente.nl>
<http://wwwtrese.cs.utwente.nl/sina>

Copyright © 1995, 1996 by TRESE, Department of Computer Science, University of Twente, Enschede, The Netherlands.

Objectworks, *Smalltalk* and *VisualWorks* are registered trademarks of ParcPlace Systems, Inc.



Contents

	1	Introduction	1
	1.1	Deficiencies in object-oriented technology	1
	1.2	The composition filters object model.	4
	1.2.1	Solving modeling problems with composition filters	5
	1.3	The past, present, and future	6
	1.4	On <i>Sina/st</i> version 3.1.	7
	1.5	On this manual	8
Part i		Development Environment	9
	2	System overview	11
	2.1	The <i>Sina/st</i> development environment	11
	2.2	The underlying Smalltalk system	12
	2.3	Basic operations	12
	3	Editing and compiling	15
	3.1	The <i>Sina/st</i> source code editor	15
	3.1.1	Menu bar menus	15
	3.1.2	Source code area	17
	3.1.3	Category area	18
	3.1.4	Compilation messages window.	18
	3.2	Your first <i>Sina/st</i> application	18
Part ii		Language Reference	21
	4	Language survey	23
	4.1	The composition filters object model.	23
	4.1.1	The inner object	24
	4.1.2	The composition filters object	24
	4.2	A composition filters object in <i>Sina/st</i>	25
	4.2.1	The interface part	26
	4.2.2	The implementation part	27
	4.2.3	The main method	29
	4.3	Forthcoming chapters	29
	4.3.1	Syntax.	29
	4.3.2	Terminology.	30
	4.3.3	Structure of a <i>Sina/st</i> program	30
	4.3.4	Where to find what	30

5	Basic language elements	31
5.1	Comment	31
5.2	Literal constants	32
5.2.1	Characters	32
5.2.2	Strings	32
5.2.3	Numbers	32
5.2.4	The constants nil, true and false	33
5.3	Variables	33
5.3.1	Declaration	34
5.3.2	Typedescription	34
5.3.3	Initial value	36
5.3.4	Assignment	36
5.4	Global externals	36
5.5	Pseudo variables	37
6	Expressions	39
6.1	Expression	39
6.2	Message expression	40
6.3	Operator expression	41
7	Control structures	45
7.1	The conditional statement	45
7.2	The for statement	45
7.3	The while statement	46
8	Classes and instances	47
8.1	Class declaration	47
8.1.1	The interface part	47
8.1.2	The implementation part	48
8.1.3	Class switches	48
8.2	An instance of a class	51
8.3	The object manager	52
9	Methods	53
9.1	Interface method	53
9.2	Private method	56
9.3	Initial method	56
9.4	Main method	57
9.5	Returning a value from a method	57
9.6	Normal versus early return	58
9.7	Method invocation	62
9.8	Statements in a method body	64
10	Conditions	65
10.1	Condition declaration	65
10.2	Condition implementation	66
10.3	Using a condition	67
11	Filters	69
11.1	Introduction	69
11.2	Input and output filters	70

11.3	Filterdeclaration	71
11.3.1	Reused filter	71
11.3.2	Local filter	71
11.3.3	Rewrite rules for filterelements	73
11.4	Filtering a message.	74
11.5	The signature of an object	76
11.6	Filterhandlers	77
11.6.1	Dispatch filter	78
11.6.2	Error filter	78
11.6.3	Send filter	78
11.6.4	Wait filter	78
11.6.5	Meta filter	79
12	Message passing semantics	81
12.1	Sending a message	81
12.2	The pseudo variable message and class ActiveMessage	83
12.3	Concurrency with threads	83
12.4	Message reification and dereification	85
12.4.1	Message reification	85
12.4.2	Message dereification	86
12.4.3	Multiple reifications	91
12.4.4	Writing ACT methods	93
12.5	The class SinaMessage	94
Part iii	Appendices	97
A	Sina/st Syntax Diagrams	99
A.1	Conventions	99
A.2	Denotations.	99
A.3	Literals	100
A.4	Program	101
A.5	Main	102
A.6	Class	102
A.7	Conditions	104
A.8	Filters	104
A.9	Methods	106
A.10	Statements	107
A.11	Expressions	108
A.12	Objects	110
A.13	Messages	111
B	Sina Class Interfaces	113
B.1	SinaObject	113
B.2	ActiveMessage	114
B.3	SinaMessage	115
B.4	ObjectManager	116
C	Using Smalltalk classes in <i>Sina/st</i>	117
C.1	<i>Sina/st</i> variable declaration using a Smalltalk class	117
C.2	Smalltalk selectors	118

References	121
Index	123

1

Introduction

This *User's Guide and Reference Manual* describes the programming language *Sina/st version 3.1* and its development environment. *Sina/st* is a Smalltalk implementation of the concurrent object-oriented programming language Sina. The Sina language incorporates the *composition filters object model* which aims at solving a number of obstacles not properly addressed by current object-oriented programming languages and software development methods. The first section of this chapter describes these shortcomings.

Some of the deficiencies of conventional object-oriented technology can be solved by the *composition filters object model*. This object model is an extension to the conventional object-oriented model, and can express several concepts, such as inheritance, delegation, synchronization, reflection on communication among objects, and real-time specifications (section 1.2)

Sina/st, Sina and the *composition filters object model* have been developed by the TRESE project¹ at the University of Twente, during several years of research. Section 1.3 gives an overview of their evolution. The final two sections of this chapter describe the features of *Sina/st version 3.1* and the global contents of this manual.

1.1 Deficiencies in object-oriented technology

The object-oriented model provides features that are very useful for a large category of applications to obtain extensible, reusable, and robust software. The conventional object-oriented model supports objects, classes, encapsulation, inheritance, part-whole relations, and polymorphic message passing [Wegner87]. The current object oriented methods, languages and tools, however, have a number of deficiencies that make them less suitable for certain categories of applications.

The TRESE project has carried out several pilot projects in order to identify the shortcomings of current object-oriented technology [Aksit92a] [Aksit95]. This research concludes that the deficiencies of the current object-oriented technology can be divided in methodological and modeling problems.

1. Twente Research & Education on Software Engineering, a research project of the SETI group at the Department of Computer Science, University of Twente, Enschede, The Netherlands.

Methodological problems

Methodological problems arise if object-oriented software development methods are utilized. The difficulties experienced in applying object-oriented analysis and design methods are addressed by other research.

Modeling problems

Modeling problems are due to the fact that object-oriented models are not capable of expressing certain aspects of applications. This category of problems is addressed by the *composition filters object model*.

We now describe these modeling problems and indicate in which application domains they can be experienced. We classify application domains into a number of basic domains. A typical application may comprise several basic domains. The basic domains include application generators, concurrency and synchronization, constraint systems, control systems, databases, distribution, knowledge processing, numerical (CAD) modeling, real-time, and user interfaces.

After the composition filters object model has been introduced (section 1.2), it is demonstrated how composition filters can solve the modeling problems. The alphabetical list of these problems and associated application domains is as follows:

- *Atomicity and inheritance* (databases and distributed systems)

Atomic actions are a mechanism to preserve consistency, for example in databases where they are called *transactions*. Due to the conventional procedure-call like semantics of transactions, they do not integrate uniformly with object-oriented concepts, especially inheritance.

- *Coordinated behavior* (control systems, databases, distributed systems, real-time applications and user-interfaces)

Coordinated behavior can be encountered for example in control systems, in which several distinct units, such as controlling algorithms, sensors and actuators, must work together in order to keep the controlled system in a consistent state.

In the object-oriented model, objects interact with each other by sending messages, where the client object transmits a message to a server object, and depending on the synchronization semantics, it either waits until the server object explicitly returns from performing its task or continues with its processing. The object-oriented models do not provide high-level mechanisms to abstract coordinated behavior among several objects since message passing semantics only involves two partner objects. To implement coordinated behavior, the application code must be spread over several entities which means that the application becomes more complex, less reusable, while its interaction semantics is more difficult to enforce and verify.

- *Duality in conception between object databases and languages* (databases)

In most of the current object-oriented language-database systems programming language and database models are still separated. There are special constructs separated from the language to access the database facilities, and often queries can only be applied on a fixed number of objects.

- *Dynamic inheritance and delegation* (all application domains)

Dynamic inheritance (dynamic delegation) means that the inheritance hierarchy (delegation structure) is not fixed, but that an object can specify a set of superclasses (delegated objects) from which it may possibly inherit (delegate to). Dynamic inheritance or delegation can be needed if an application must be able to adapt for performance reasons or space requirements, to different clients or contexts, or even to different hardware, as for example in distributed systems.

- *Excessive class declarations (non-parameterized hierarchies)* (all application domains)

Excessive class declarations can be experienced when all meaningful combinations of classes must be composed with (multiple) inheritance and declared as a separate class.

Because every possible combination must be represented explicitly in the inheritance hierarchy, this hierarchy easily becomes unmanageable.

Consider for example a graphical application which needs to draw points on a display. The class `Point` represents such a display point. Class `HistoryPoint` inherits from `Point` and keeps a list containing the sequence of locations the point has had. Class `BoundedPoint` has restricted display coordinates and also inherits from `Point`. Class `BoundedHistoryPoint` combines the features of `HistoryPoint` and `BoundedPoint` and therefore inherits from both these two classes.

Assume now that, for the drawing of lines, we want to define class `SolidLine` which inherits from `Point`. `DashedLine` inherits from `SolidLine` but draws dashed lines instead of solid lines. If all the combinations of these classes are meaningful then one needs to declare all the possible combinations as a separate class, such as `Point`, `HistoryPoint`, `BoundedPoint`, `BoundedHistoryPoint`, `SolidLine`, `DashedLine`, `HistorySolidLine`, `BoundedSolidLine`, `BoundedHistorySolidLine`, `HistoryDashedLine`, `BoundedDashedLine`, `BoundedHistoryDashedLine`. The inheritance hierarchy will grow further if we add more classes to the hierarchy. For example, creating class `3DimensionalPoint` as a subclass of `Point` doubles the number of class declarations.

- *Fixed message passing semantics* (concurrency and synchronization, user interfaces)

Although message passing is a key feature in the object-oriented model, the semantics of message passing is fixed by the adopted language, or, at best, can be selected from a limited set of fixed semantics. Message passing semantics may include remote procedure call mechanism, asynchronous message passing, broadcast and multicasting of messages, and atomic message send semantics. In most languages, these fixed semantics cannot be tailored to model application specific interaction mechanisms, while it is even harder to abstract and reuse them.

- *Inheritance versus real-time* (real-time systems)

When classes are reused in applications with different real-time behavior, changes to either the application requirements or the real-time specifications in subclasses may result in excessive redefinitions of superclasses while this may be intuitively unnecessary. This so called *real-time specification anomaly* can arise when real-time specifications have been mixed within the application code, when real-time specifications are not polymorphic (i.e. they cannot be used for more than one method), or when real-time specifications are orthogonally restricted (i.e. independently defined specifications cannot be combined through inheritance without redefinition of the related specifications).

- *Inheritance versus synchronization* (concurrency and synchronization, user interfaces)

Conflicts between inheritance and synchronization can appear when a class that implements synchronization is extended in a subclass by introducing new methods, overriding methods, or by adding synchronization constraints. When this requires additional redefinitions, this problem is considered to be an *inheritance anomaly*².

For instance, when synchronization code is mixed with the application code (i.e. inside a method), changing synchronization is impossible without affecting the application code. Another source for inheritance anomalies comes from the non-decomposability of synchronization specifications in some languages. It is thus hard to add a new synchronization constraint, or to redefine a part of the synchronization specification in a subclass.

- *Multiple interfaces (views)* (all application domains)

Not all operations provided by an object are necessarily of interest to other objects that use its services. Therefore it is desirable to define interfaces for an object, differentiating between clients, that is, between the senders of a message. For example, a public mailbox should make a distinction between a postman and others, since

2. This term is not restricted to synchronization. If the introduction of a new method or overriding an inherited method in a subclass requires additional redefinitions, this problem is an inheritance anomaly, because this should not be necessary.

everybody is allowed to put a letter in it, but only a postman is allowed to empty the mailbox.

- *Part-of versus inheritance* (all application domains)

A part in the analysis phase may be implemented by inheritance in the design phase, which means that there are conceptual differences between the analysis phase and design phase and this may result in changes in the object oriented model of the application. As an example, consider a rectangle. Conceptually, a rectangle can be described with two points, i.e. they are part of the rectangle. If we would like to emphasize reusability, a rectangle could be defined to inherit from class Point instead.

- *Lack of reflection* (all domains, but typically in control systems, distributed systems, knowledge processing, and user interfaces)

A reflective system is a system which incorporates models representing (aspects of) itself. This self representation is causally connected to the reflected entity, and therefore, makes it possible for the system to answer questions about itself and support actions on itself. Conventional object-oriented methods and languages do not, or only provide a limited support for reflection.

- *Sharing behavior with state* (all application domains)

In conventional object-oriented models, classes cannot conveniently express sharing behavior that is affected by shared state information. This is because, in general, classes are used to share behavior while their instances store their respective states. Sharing behavior with state is needed, for instance, when we want to model a company with several departments. Every department has a secretary who takes care of making appointments (shared behavior), and maintains the common agenda (shared state) of the department employees. When an employee is asked to make an appointment, he will delegate this request to the secretary who takes the agenda and fixes a time and date based on the employee's working hours.

- *Unmatched system functionalities* (all but specifically in constraint systems, databases, and distributed systems)

The operating system generally provides services that are required by most applications. However, some applications may require specific functionalities which are not supported by it. These then, must be implemented by the software engineer. For example, only a few operating systems provide recovery mechanisms needed by a database application after a transaction has failed.

1.2 The composition filters object model

The *composition filters object model* extends the basic object-oriented model in a modular way. Its object model can be divided in two separate parts: an implementation part and an interface part. The implementation part is the kernel of an composition filters object and adheres to the conventional object-oriented model. The interface part is a layer that encapsulates the kernel object. This part contains the extensions which are made by the composition filters model. The interface part receives incoming messages, processes them, and optionally hands them over to the implementation part. The interface part additionally processes messages that are sent by the implementation part, and sends them to their respective targets.

The extensions that are introduced by the composition filters object model comprise *input* and *output composition filters*. Input filters process received messages, while output filters process sent messages. Every filter has its own functionality which does not affect other filters: filters are orthogonal. Filters can be *composed* in arbitrary combinations. The combined functionalities of these filters are thus added to the kernel object (the implementation part).

A filter specifies in a declarative way which messages it accepts. Whether the filter accepts a message depends on the message itself (typically its selector), and some designated

conditions. These conditions can access the context of the message (e.g. its sender), and the state of its receiver.

The composition filters object model defines the following filters:

- *dispatch filter*: a dispatch filter offers a message to a specified target object for further processing. Conceptually, a dispatch filter realizes inheritance when the target object is encapsulated by (i.e. internal to) the receiver, and delegation when it is external to the receiver. In the former case, the target object represents the superclass. Associative inheritance and delegation³ is accomplished when conditions—which reflect the context or state of the receiver—are used to choose between several target objects.
- *wait filter*: a wait filter is used to delay a message. Conditions specify the synchronization constraints. The message is allowed to continue when the synchronization conditions are met;
- *meta filter*: a meta filter suspends a message and offers it to a designated object. This object, an *abstract communication type* (ACT), manipulates the message and releases it again. A meta filter allows an ACT to reflect on communication among objects;
- *error filter*: an error filter raises an error exception for a message;
- *real-time filter*: a real-time filter is able to affect timing constraints of a message;
- *send filter*: a send filter offers a sent message its target.

1.2.1

Solving modeling problems with composition filters

The object-oriented modeling problems described in section 1.1 can be solved by using the composition filters object model. In this section we indicate how.

- *Atomicity and inheritance*;

Atomic messages have been introduced to support atomicity. Because they can be specified by every filter, atomic inheritance and delegation is provided for as well [Aksit91]. (this subject is not addressed in this manual)

- *Coordinated behavior*;

An abstract communication type object can be used to coordinate the behavior among the group of objects. Messages between a pair of objects in this group can be intercepted by a meta filter, and offered to the coordinating object. Because the application code implementing the coordinated behavior has not been divided over the participating objects, it can be easily extended or replaced [Aksit93]. (refer to section 11.6.5, page 79 and section 12.4, page 85)

- *Duality in conception between object databases and languages*;

Database-like features can be accomplished by a dispatch filter, associative inheritance, and set operations. There is no separate language construct for these features, while queries can be applied to all objects [Aksit92b]. (not addressed in this manual)

- *Dynamic inheritance and delegation*;

This problem can be solved by using dispatch filter and associative inheritance c.q. delegation. Because conditions representing the context or state of the receiver are evaluated dynamically, the inheritance and delegation structure changes appropriately [Aksit92b]. (refer to section 11.6.1, page 78 and section 11.6.2, page 78)

Behavior injection

Alternative implementations can be realized with *behavior injection*. Behavior injection is defined as replacing or extending the behavior of an object dynamically (i.e. at run-time) with new behavior. In the composition filters object model this can be accomplished by replacing an object which is the target in a dispatch filter, with another object.

3. By using associative inheritance and associative delegation, the client may affect the inheritance hierarchy, c.q. delegation structure to some extent.

- *Excessive class declarations (non-parameterized hierarchies);*

The dispatch filter and associative inheritance can be used to avoid excessive class declarations. A client can choose the right combination of superclasses from the given ones by setting the appropriate conditions at instance creation time [Aksit92b]. (refer to section 11.6.1, page 78)

- *Fixed message passing semantics;*

By using meta filters and abstract communication types, we can introduce arbitrary message passing semantics. Every abstract communication type object can implement specific message passing semantics, such as asynchronous and future message passing. Because these abstract communication type objects are instances of ordinary classes, message passing semantics can be reused and extended [Aksit93][Bergmans94a]. (refer to chapter 12)

- *Inheritance versus real-time;*

The real-time filter can change the timing constraints of a message. Since real-time filters separate real-time specifications from methods, the real-time specification anomalies are eliminated [Aksit94]. (not addressed in this manual)

- *Inheritance versus synchronization;*

A wait filter in combination with conditions can be applied to specify synchronization constraints. By using wait filters, the synchronization inheritance anomalies are avoided [Bergmans94a]. (refer to section 11.6.4, page 78)

- *Multiple interfaces (views);*

Multiple interfaces can be specified by using a dispatch filter or error filter with conditions based on the context. These conditions are used to differentiate between (groups of) clients [Aksit92b]. (refer to section 11.6.1, page 78 and section 11.6.2, page 78)

- *Part-of versus inheritance;*

This problem can be avoided by using a dispatch filter and an internal as superclass. The composition filters object model does not make a distinction between part-of and inheritance relation: conceptually there is an inheritance relation when an internal object is the target object of a dispatch filter [Aksit88][Aksit92b]. (refer to section 4.2.1, page 26)

- *Lack of reflection;*

Reflection on messages and the communication between objects is supported by the meta filter and abstract communication types [Aksit93]. (refer to section 11.6.5, page 79 and section 12.4, page 85)

- *Sharing behavior with state;*

This problem can be solved by applying a dispatch filter which delegates a message to an object external to the receiver. As this object is not encapsulated by the receiver, it can be shared by several other objects. Since this shared object is an instance of some class, it provides common behavior while it encapsulates the shared state [Aksit88][Aksit92b]. (refer to section 4.1.2, page 24 and section 11.6.1, page 78)

- *Unmatched system functionalities;*

Abstract communication type objects, together with meta filters can be used to implement extensible system functionalities [Aksit93]. (refer to section 11.6.5, page 79 and section 12.4, page 85)

1.3 The past, present, and future

The composition filters object model evolved from earlier versions of the programming language Sina. The Sina language was named after the medieval philosopher, scientist and physician Ibn Sina who is also known under the Latin name of Avicenna.

The first version of Sina was developed around 1985 by Mehmet Aksit when he was working at the Research department of Océ Nederland B.V. A preliminary version of the composition filters concept was called *semantic network*. This semantic network construction served as an extension to objects (classes, messages, instances) which could be *configured* to form other objects, such as classes from which instances could be created. An object manager took care of synchronization and message processing of an object. The semantic network construction already could express the key concepts of delegation, reflection, and synchronization.

In 1987, when Mehmet Aksit moved to the University of Twente, the semantic network concept evolved to *interface predicates* which describe which messages an object is able to handle. In the second version of Sina, the functionality of the object manager was reduced, and the possibility to configure objects into classes was abolished. To abstract communications among objects, the notion of *abstract communication type* was introduced, while atomic delegations made it possible to describe transactions [Aksit89]. In this period, Anand Tripathi of the University of Minnesota contributed to the language design, while Hans Bank, Ruud Nijhuis, Jan-Willem Dijkstra, Willem Veldkamp, Gerard van Wageningen, and Nini Poortenga were involved as students.

The TRESE project was established by Mehmet Aksit in the beginning of 1991. As result of ongoing research, the foundation for the present day composition filters object model was defined in the same period. The interface predicates of the prior version were replaced by the dispatch filter and the wait filter took over the synchronization functions of the object manager. Message reflection and real-time specifications are handled by the meta filter and the real-time filter, respectively. The composition filters object model turned to be the major focus of attention of the TRESE project while the language Sina became an expressive instrument for it. As a result, the third version of Sina, *Sina/st version 3.1*, integrates the composition filters object model.

Contributions to the composition filters object model and the Sina language were made by the project members Lodewijk Bergmans, Richard van de Stadt, Jan Bosch, Ken Wakita, Sinan Vural, Enis Yücesoy and Sander Pool, and the students Koen Lesterhuis, William van der Sterren, Piet Koopmans, Coen Stuurman, Maurice Glandrup, John Mordhorst, and Wietze van Dijk.

The composition filters object model already has been applied in several projects. These include an atomic transactions framework [Tekinerdogan94], a process programming framework [Algra94], an image algebra framework [Vuijst94], and a fuzzy-logic reasoning framework [Marcelloni95].

Currently, the TRESE project is extending the conventional object-oriented languages Smalltalk [Goldberg83][Goldberg89] and C++ [Ellis90] with composition filters. This is possible because the composition filters object model is a modular extension to the basic object-oriented model. Further, some practical improvements to the syntax and semantics of composition filters have been introduced, while research continues on new filter types, such as the active filter, the multiple-dispatch filter, the atomic filter, and the substitution filter.

1.4 On *Sina/st version 3.1*

Sina/st version 3.1 is a Smalltalk implementation of the concurrent object-oriented programming language Sina. It has been developed using *VisualWorks* and needs ParcPlace's *VisualWorks* or *VisualWorks* as execution environment. *Sina/st* includes a kernel that gives run-time support, and a compiler which transforms Sina code into Smalltalk. Smalltalk classes and objects can be used within *Sina/st* as if they are Sina classes c.q. objects.

Sina/st version 3.1 incorporates the composition filters object model by providing the following composition filters:

- dispatch filter;
- error filter;
- send filter;
- wait filter;
- meta filter.

Concurrent activities can be started by using a so called *early return* in a method. This allows the method to execute statements concurrently with the caller after it has returned an object to it.

Note that not all the concepts mentioned in section 1.2 are available in *Sina/st version 3.1* yet. Atomic messages, database and transaction functionality, and the real-time filter are not provided. Also only a limited form of type checking has been implemented.

1.5 On this manual

The *Sina/st User's Guide and Reference Manual* provides a comprehensive description of the *Sina/st* development environment and the *Sina/st* language. It is divided into three parts:

- Part I: *DEVELOPMENT ENVIRONMENT*

This part describes the development environment, including the user interface which is used to compile and execute *Sina/st* code;

- Part II: *LANGUAGE REFERENCE*

This part describes the *Sina/st* language in full detail. Its first chapter (page 23) provides a language survey of the language including a description of the composition filters object model and an index to the following chapters;

- Part III: *APPENDICES*

This part contains relevant information, including *Sina/st* syntax diagrams and class interface descriptions.

Further reading

You can find a detailed description of the programming language Smalltalk, the language which can be used in *Sina/st*, in the next two books:

- *Smalltalk-80: The Language And Its Implementation* [Goldberg83];
- *Smalltalk-80: The Language* [Goldberg89].

The following manuals provide further information on the *VisualWorks* and *Objectworks\Smalltalk* programming environments:

- The *Objectworks\Smalltalk User's Guide* [ParcPlace92a];
- *VisualWorks User's Guide* [ParcPlace92b].



PART i

Development Environment

2

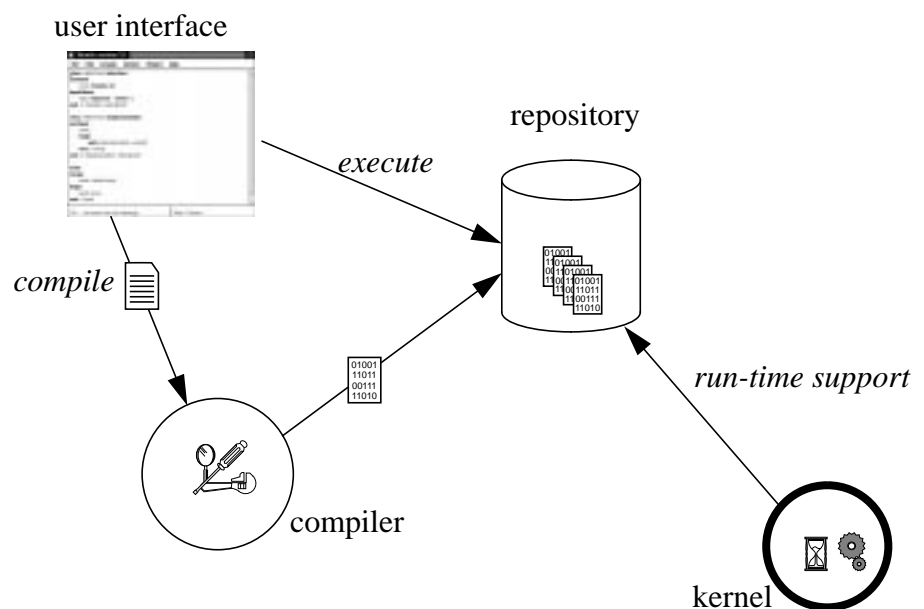
System overview

This chapter gives an overview of the *Sina/st* development environment. As the *Sina/st* system resides in the *VisualWorks* environment, it also addresses the underlying Smalltalk system (section 2.2). The last section describes how to install, start and save the *Sina/st* and Smalltalk environment, and how to operate the mouse buttons.

2.1 The *Sina/st* development environment

The *Sina/st* development environment consists of four main components which are shown in figure 2-1.

FIGURE 2-1 *Sina/st* development environment



A *Sina/st* programmer will mainly interact with the *user interface*. From there, the user can write and edit *Sina/st* source code, compile and execute it. An application in *Sina/st* is composed of a number of classes and a main method to start the application. The *compiler* verifies the syntax and semantics of the *Sina/st* source code. If this is correct, it will generate code which is stored in the *repository*. The *repository* is a collection of objects

SinaCompilerInterface
SinaSystemDictionary

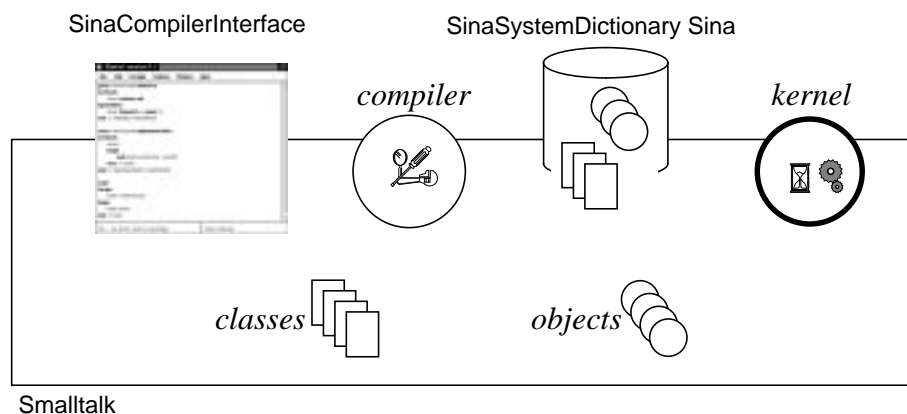
and predefined or newly compiled classes which can be used in an application. From the *user interface*, you can start the execution of an application. The *kernel* provides run-time support for this, such as message filtering and process scheduling.

The *user interface* is represented by the object called SinaCompilerInterface. The following chapter describes how to open the *user interface* and how to edit, compile and execute *Sina/st* code within it. The object SinaSystemDictionary called Sina embodies the *repository*. An application can create objects in and store them in the *repository*, while the *compiler* places compiled classes in it. The *compiler* and the *kernel* exist as a collection of classes and objects which are normally invisible to the user.

2.2 The underlying Smalltalk system

The *Sina/st* development systems resides in the *VisualWorks* environment. *VisualWorks* includes a system dictionary, called Smalltalk, containing Smalltalk classes and objects¹. Smalltalk classes and objects are available in an executable form.

FIGURE 2-2 *Sina/st in the Smalltalk environment*



The entire *Sina/st* development system has been written in Smalltalk and is composed of a set of –Smalltalk– classes and objects (figure 2-2). The *Sina/st* compiler compiles a Sina class to a Smalltalk class. The SinaSystemDictionary Sina is extension to the Smalltalk system dictionary: it contains Sina objects and compiled Sina classes. Through the SinaSystemDictionary Sina, existing Smalltalk classes and objects in the Smalltalk system dictionary can be accessed. Smalltalk classes and objects can be used in a *Sina/st* application as if they are normal Sina classes cq. objects.

2.3 Basic operations

In this section we will briefly describe how to manage the *Sina/st* and Smalltalk environment. For detailed information on Smalltalk, consult your *VisualWorks User's Guide* [ParcPlace92a] or *VisualWorks User's Guide* [ParcPlace92b].

Starting Smalltalk

VisualWorks is available for a wide variety of computer systems, under several window systems. Starting *VisualWorks* requires a different procedure for each of them. For systems where you can start *VisualWorks* by entering a shell command line, you should enter a command of the following form:

```
vmname imagename
```

For vmname, substitute the name of the virtual machine (this is `visualworks` on UNIX and Macintosh systems, and `vw.exe` on PC systems). For imagename, substitute the

1. In Smalltalk, every class is an object. A class is able to create new objects.

	name of the standard image (<code>st80.im</code> or <code>visual.im</code>) or the name of an image in which the <i>Sina/st</i> development environment already has been installed.						
<i>Saving an image</i>	<p><i>VisualWorks</i> allows you to save the complete environment, including the <i>Sina/st</i> system, in a so-called <i>image</i>. You can restart Smalltalk at a later moment with this image file and continue with your environment as you saved it.</p> <p>There are two ways of saving an image. The first is to select the Save As... from the File menu in the <i>VisualWorks</i> main window. The second possibility to save an image is upon leaving <i>VisualWorks</i>: choose Exit VisualWorks... from the File menu in the Launcher window, then select the Save then Exit option. In both cases you will be prompted for an image name, with the current image name as default.</p>						
<i>Installing the Sina/st development system</i>	<p>To install the <i>Sina/st</i> development environment in a Smalltalk image, you should 'file in' the necessary classes. The <i>Sina/st</i> package comes with file called <code>Install.st</code> (or <code>install.st</code> in the PC version of the package) which performs this task. Once <i>Sina/st</i> has been installed, the image can be saved: this image can be used at a later time, without having to install <i>Sina/st</i> again.</p> <p>You should open a File List on the directory named <code>src</code> which contains file the <code>Install.st</code> (or <code>install.st</code>). To open a File List, select File List from the Tools menu. Choose the file called <code>Install.st</code> or <code>install.st</code> in the File List, and follow the instructions in it. After a while, <i>Sina/st</i> will have been installed and you can save the image as described above. To open the <i>Sina/st</i> user interface, read the next chapter.</p>						
<i>Operating the mouse</i>	<p><i>VisualWorks</i> uses three mouse buttons to perform various functions described below. They are called <select>, <operate> and <window>. These three buttons are usually mapped to the left, middle and right button on the mouse, but on hardware configurations with a one-button (Macintosh) or a two-button mouse (PC), the 'missing' buttons can be accessed by pressing a mouse button in combination with an appropriate key: refer to your Smalltalk <i>User's Guide</i> for this.</p> <table> <tr> <td><select> button</td><td>This button selects a window location or menu item, positions the textcursor, or highlights text while you drag the mouse.</td></tr> <tr> <td><operate> button</td><td>This button brings up a menu of operations that are appropriate for the current view or selection. This menu is referred to as the <operate> menu.</td></tr> <tr> <td><window> button</td><td>This button brings up a menu of actions that can be performed on any <i>VisualWorks</i> window, such as move and close.</td></tr> </table>	<select> button	This button selects a window location or menu item, positions the textcursor, or highlights text while you drag the mouse.	<operate> button	This button brings up a menu of operations that are appropriate for the current view or selection. This menu is referred to as the <operate> menu.	<window> button	This button brings up a menu of actions that can be performed on any <i>VisualWorks</i> window, such as move and close .
<select> button	This button selects a window location or menu item, positions the textcursor, or highlights text while you drag the mouse.						
<operate> button	This button brings up a menu of operations that are appropriate for the current view or selection. This menu is referred to as the <operate> menu.						
<window> button	This button brings up a menu of actions that can be performed on any <i>VisualWorks</i> window, such as move and close .						

3

Editing and compiling

In this chapter we describe the *Sina/st* source code editor (section 3.1), and in order to get the feeling with it, we write our first *Sina/st* application (section 3.2).

3.1 The *Sina/st* source code editor

By using the *Sina/st* source code editor, it is possible to write, compile and execute Sina programs. A source code editor window can be opened by evaluating the following Smalltalk expression¹.

```
SinaCompilerInterface open
```

A window similar to that of figure 3-1 will appear (the actual appearance can differ depending on the windowing system you are using). In this window we can see the following four areas:

- the menu bar allows you to perform commands for saving, editing and compiling Sina code (section 3.1.1);
- the source code area, in which Sina code can be typed and edited (section 3.1.2);
- the status line shows what the compiler is doing;
- the category area determines the location where compiled code is stored (section 3.1.3).

The compilation messages window, described in section 3.1.4, displays errors, warnings and informative messages resulting from compiling Sina source code. The compilation messages window can be opened from the editor's **Window** menu. More than one editor window can be opened, each of which has its own compilation messages window.

3.1.1 Menu bar menus

Almost all the operations for editing, compiling, and running Sina programs can be performed from menus of the menu bar. These menus are described in this section.

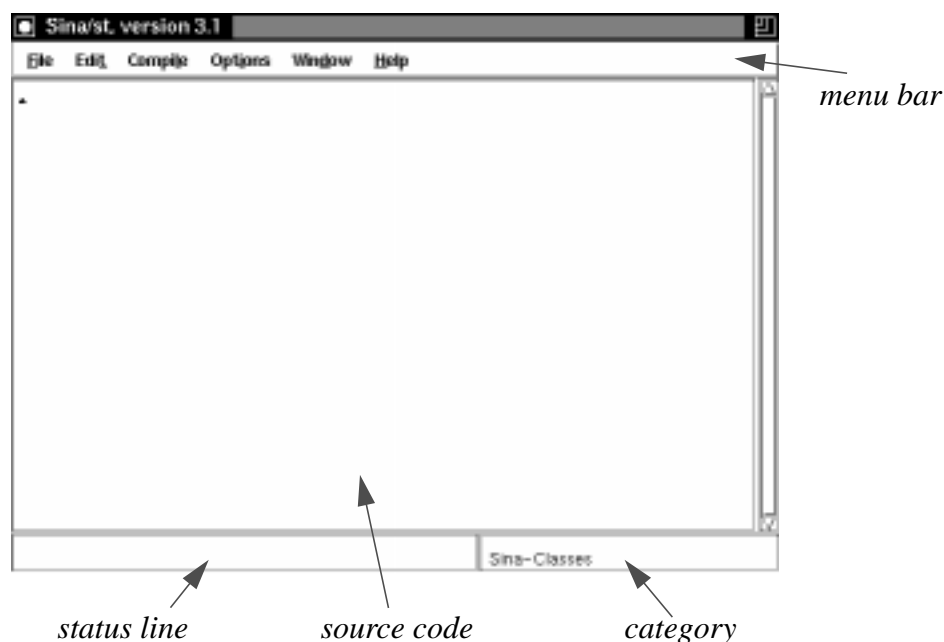
File menu

The **File** menu has the following commands.

1. To evaluate the expression, type the expression in a Smalltalk text window, for instance the Transcript window, and select the text with the <select> button. Then choose **do it** from the <operate> menu.

Open...	Prompts for a filename. Opens the file and displays the source code. If the Auto Format option (see Options menu) is enabled, the code will be formatted.
Save	Save the source code in the currently opened file. If no file was opened, it prompts for a filename.
Save As...	Prompts for a filename and saves the source code in the specified file.
File Browser	Opens a File Browser.
Edit Class...	Prompts for the name of a class. Loads the source code of this class.
Hardcopy	Prints a copy of the source code on paper.
Exit	Closes the window.

FIGURE 3-1

User interface**Edit menu**

The **Edit** menu has the following commands.

Undo	Revokes the most recent cut or paste.
Cut	Places a copy of the highlighted text in the paste buffer, then deletes the original.
Copy	Places a copy of the highlighted text in the paste buffer.
Paste	Deletes the highlighted text (if any), then places a copy of the most recent copied or cut selection in that location.
Find...	Prompts for a text, and search for it.
Replace...	Prompts for a text and a replacement, then searches the text and replaces it.
Format	Pretty prints the source code.
Class template	Inserts a template representing the structure of a Sina class.
Main template	Inserts a template representing the structure of main .

Compile menu

The **Compile** menu has the following commands.

Compile	Compiles the source code. No compilation will take place if the source code has not changed since it was last compiled. However, by pressing <shift> while selecting this command, compilation is forced.
Run	Compiles the source code if necessary. If the source code contains a main method, it is executed.

Options menu

The **Options** menu enables or disables the following options.

Provide default behavior Provide each compiled class with default behavior by inserting the default object and default behavior filter (see page 49).

Provide mutual exclusion Provide each compiled class with mutual exclusion by inserting the default synchronization filter (see page 49).

Confirm code regeneration Confirm when the (Smalltalk) code of a class is regenerated.

Confirm source code formatting Confirm when the source code is formatted.

Auto format source code Format the source code upon opening a file (**File→Open...**) or editing a class (**File→Edit Class...**).

Window menu

The **Window** menu has the following commands.

- Sina** Inspects the SinaSystemDictionary Sina. A window similar to that of figure 3-2 (left) appears. In the left view, the names of objects, typically externals, appear. By selecting one of them, its value is displayed in the right view.
- Messages** Opens the compilation messages window (see section 3.1.4).
- New** Opens a new source code editor window.
- Spawn** Opens a new source code editor window containing the same source code.

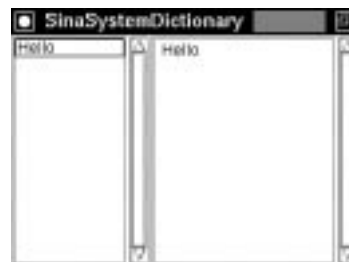
Help menu

The **Help** menu has the following commands.

- About Sina...** Displays information on Sina.

FIGURE 3-2

The SinaSystemDictionary window



3.1.2 Source code area

In the source code area of the *Sina/st* source code editor, Sina code can be edited. The **<operate>** menu in this area has the following commands (which are also available from the menu bar menus):

- find...** Prompts for a text, and search for it.
- replace...** Prompts for a text and a replacement, then searches the text and replaces it.
- undo** Revokes the most recent cut or paste.
- copy** Places a copy of the highlighted text in the paste buffer.
- cut** Places a copy of the highlighted text in the paste buffer, then deletes the original.
- paste** Deletes the highlighted text (if any), then places a copy of the most recent copied or cut selection in that location.
- cancel** Undo all edits and revert to the code as it was compiled the last time.
- format** Pretty prints the source code.
- messages** Opens the compilation messages window (see section 3.1.4).
- run** Compiles the source code if necessary. If the source code contains a **main** method, it is executed.
- compile** Compiles the source code. No compilation will take place if the source code has not changed since it was last compiled. However, by pressing **<shift>** while selecting this command, compilation is forced.

3.1.3 Category area

In the category area you can enter the class category in of the Smalltalk system dictionary in which the compiler stores a compiled Sina class. However, if a category has been specified in the source code of a class with the class switch `#Category` (refer to section 8.1.3 on page 48), the compiled class will be stored in the designated category. The **<operate>** menu has the following options:

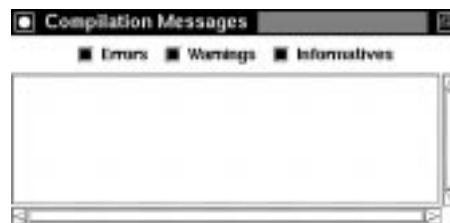
copy	Place a copy of the highlighted text in memory.
cut	Place a copy of the highlighted text in memory, then delete the original.
paste	Delete the highlighted text (if any), then place a copy of the most recently copied or cut selection in that location.
accept	The text in this view becomes the new category. This has the same effect as ending the input by hitting the return key.
cancel	Restore the text into its state before it was last changed.
select...	Prompts to select one of the available categories.

3.1.4 Compilation messages window

The compilation messages window displays errors, warnings and informative messages resulting from compilation (figure 3-3). This window can be opened, or brought to front, by selecting the source code editor's **Window→Messages** menu command.

FIGURE 3-3

The compilation messages window



The top buttons in the compilation messages window allow you to selectively display or hide error, warning and informative messages. The **hardcopy** option of its **<operate>** menu prints a copy of the messages on paper.

The following error message is an example of a compilation message that could appear in the compilation messages window.

Error- T:foo - Smalltalk class SmallInteger does not understand message bar -> 4.bar

A compilation message shows

- the kind of message: an error, as in this case, or a warning or informative;
- where it was found: in the method foo of class T;
- what is wrong;
- and the offending statement: 4.bar.

3.2 Your first Sina/st application

To get accustomed with all of this, we can write our first *Sina/st* application. It is the wellknown salute to our planet. You can type the code of example 3-1 in the source code area of a newly opened source code editor.

Now, you are ready to compile it. In the source code area, bring up the **<operate>** menu and select the **compile** option. To show the compilation messages select **messages** from the **<operate>** menu. Alternatively, you can select **Compile→Compile** and **Window→Messages** from the menu bar.

EXAMPLE 3-1

```

class HelloWorld interface
  methods
    show returns nil;
  inputfilters
    disp: Dispatch = {inner.*};
end;

class HelloWorld implementation
  methods
    show
      begin
        self.printLine("Hello, world!");
      end;
end;

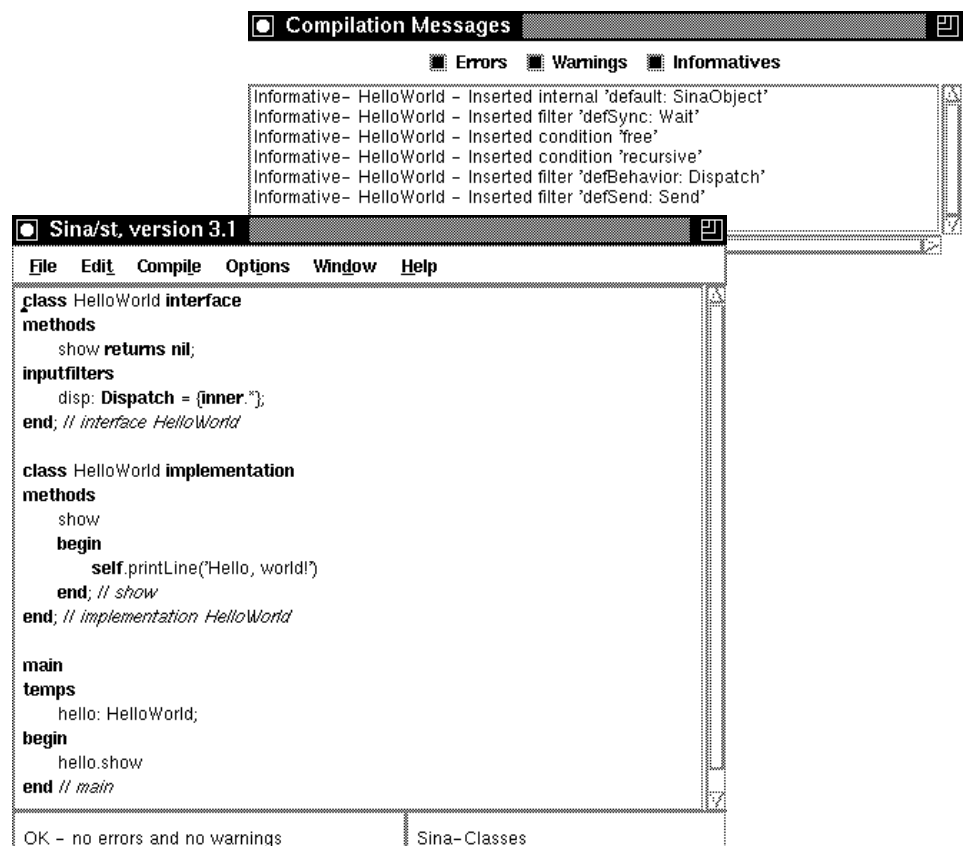
main
  temps
    hello : HelloWorld;
  begin
    hello.show
  end
end

```

Figure 3-4 shows the result of the compilation. Provided that you typed the source code correctly, the status line of the editor reports that it has generated code and that it has found no errors or warnings. In the compilation messages window only informative messages are present: the compiler has added an internal object, two conditions and three filters to the class HelloWorld. They are inserted automatically and provide the class with default behavior and synchronization.

FIGURE 3-4

Compilation of HelloWorld

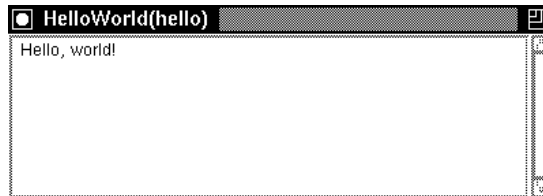


In the **main** method, an instance of class HelloWorld is created and asked to show its message. To execute **main**, choose the **run** option from the <operate> menu. The window

of the object `hello` (figure 3-5) will emerge: its title shows that it is the window of the object `hello` and that this object is an instance of the class `HelloWorld`. If you run **main** again, another instance of `HelloWorld` will be created that has its own window. To close an object's window, bring up the **<window>** menu with the **<window>** button and select the **close** option. Then, answer yes to the question to discard the information in the window.

FIGURE 3-5

The window of the object hello



You could try the following main method instead of that of example 3-1, or use your own creativity.

```
main
  temps
    hello : HelloWorld;
  begin
    hello.show;
    hello.println(3+4);
    hello.println(hello.request("What is your name?"));
    hello.warn('And now an error');
    hello.foo
  end
```



PART ii

Language Reference

4

Language survey

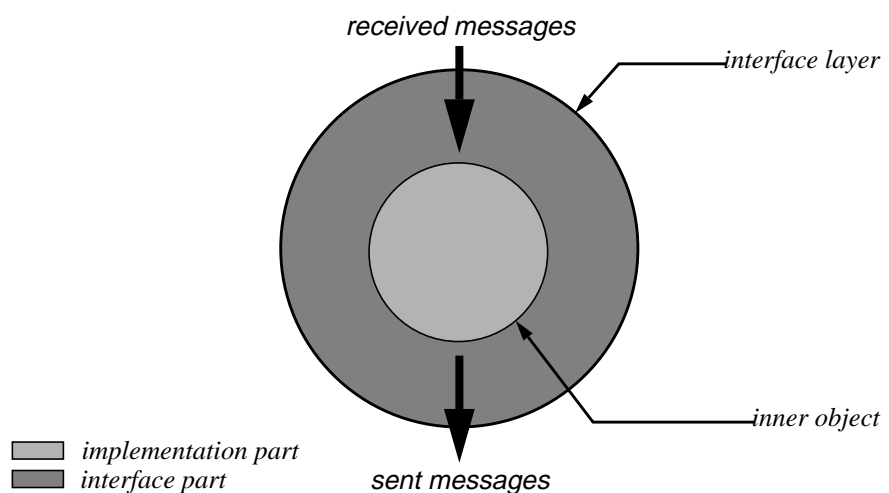
In this chapter, we explain the composition filters object model in short and give an overview of the language elements of *Sina/st*. The last section outlines the rest of the manual.

4.1 The composition filters object model

The composition filters object model is class based: for an application consisting of a number of objects, all objects with common characteristics are instances of the same class. Objects are only specified on class level; this supports the creation of multiple instances and reuse through the inheritance mechanism ([Bergmans94a][Bergmans94b]).

FIGURE 4-1

An object composed of an interface layer and an inner object



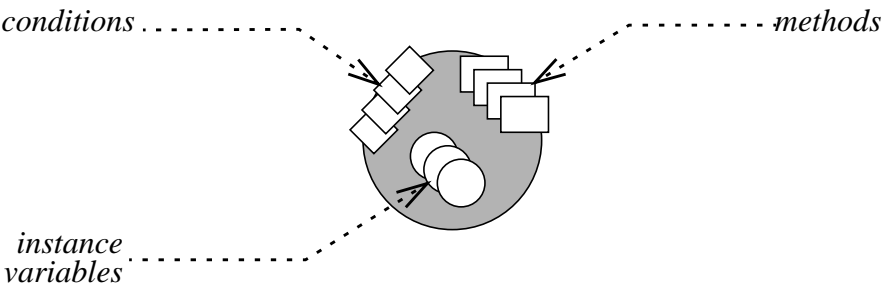
As can be seen in figure 4-1, an object in the composition filters object model is composed of an inner object and an interface layer. The inner object implements the object's specific behavior. The interface layer encloses the inner object and manages incoming and outgoing messages.

In *Sina/st* the interface layer is defined by the *interface part* (section 4.2.2), and the inner object by the *implementation part* (section 4.2.1). The coming section describes the inner object, while section 4.1.2 covers the complete composition filters object.

4.1.1 The inner object

The inner object implements the specific behavior of an object and holds its state. As pictured in figure 4-2, the inner object contains three main components: *methods*, *conditions* and *instance variables*. The names of the methods and the conditions are visible on the encapsulating boundary of the inner object, whereas the instance variables are fully encapsulated by it. The inner object is embodied in *Sina/st* by the implementation part and is explained in section 4.2.2.

FIGURE 4-2 The inner object with instance variables, methods and conditions



Instance variables	An instance variable holds the state of an object. An instance variable is an instance of a specific class which is created when the inner object is created.
Methods	The behavior of an object is implemented by its methods. A method defines a number of actions that are performed in reaction to the invocation of the method.
Conditions	A condition provides information about the current state of the object. It is a special kind of method that takes no parameters and returns a boolean result.

4.1.2 The composition filters object

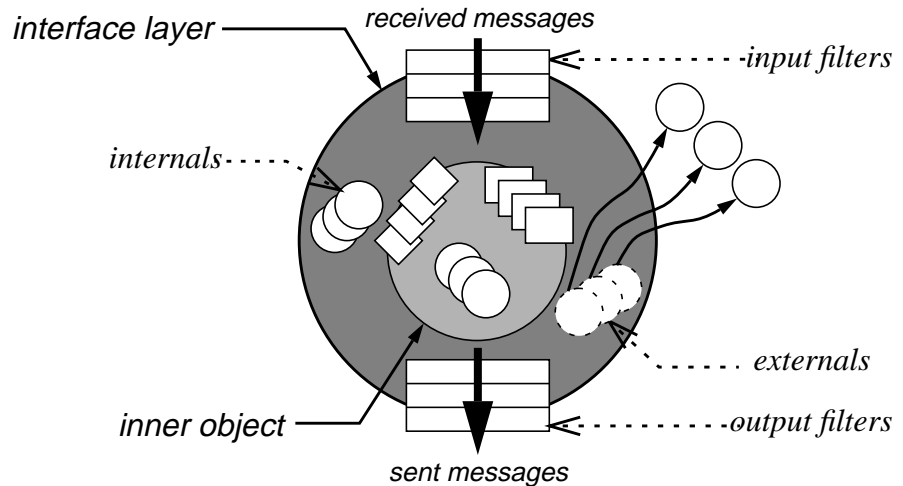
Figure 4-3 shows the inner object being surrounded by an interface layer. The following components can be distinguished in the interface layer: *internals*, *externals*, *input filters* and *output filters*. The interface layer is embodied in *Sina/st* by the interface part and is explained in section 4.2.1.

Internals	Internals are fully encapsulated objects and are used to compose the behavior of the composition filters object: a received message could be delegated to an internal object instead of the inner object, for instance.
Externals	External objects are objects that exist outside the composition filters object, such as global objects. They can be used to share data among objects (<i>sharing behavior with state</i>). An example of this is a department secretary which is an external for all the managers of this department. When a manager is asked to make an appointment, he delegates this request to the secretary which maintains his agenda. Similarly to internals, they are also used to compose the behavior of the composition filters object. In figure 4-3 the interior structure of the internal and external objects and instance variables is not shown, but they are composition filters objects as well, having an inner object and an interface layer.
Input filters	The input filters handle messages that are received by the object. One type of filter that can be part of the input filters is the so called <i>dispatch filter</i> which can either invoke a method of the inner object, or offer the message to the input filters of an internal or external object.
Output filters	The output filters handle messages that are sent by the object. A so called <i>send filter</i> delivers the message at the input filters of the receiver of the message.

Sending a message

When an object (the *sender*) sends a message to another object (the *receiver*), the message, in general, is filtered by the output filters of the sender, and then by the input filters of the receiver. Every filter decides what to do with the message. Eventually, there is some filter (a dispatch filter) that invokes a designated method. Hence, filtering a message can be thought of as the method lookup.

FIGURE 4-3

The composition filters object*Filtering a message*

When a filter filters a message, the filter either accepts or rejects the message. If a dispatch filter accepts a message, it invokes a method or offers the message to the input filters of an internal or external. A rejected message continues in the following filter. Similarly, a send filter delivers the message to the input filters of the receiver when it accepts a message, and offers it to the next filter when it rejects a message.

Filtertypes

Filtertypes that can appear in the input filters or output filters group include the aforementioned *dispatch filter* and *send filter*, and further the *wait filter*, the *error filter* and the *meta filter*. The wait filter can be used for inter and intra object synchronization: when it rejects a message, the execution of the message is suspended. The execution resumes when a certain condition is met, causing the message to be filtered by the following filter. The error filter raises an error when it rejects a message. The meta filter is able to reflect on message communication between objects. When a message is accepted by this filter, it is offered to another object. This object is thus able to reflect on the message, i.e. the communication between two objects.

In the class interface part, the input filters and output filters can be declared containing an arbitrary set of those filters. In this way, the behavior of an object can be composed by selecting the appropriate combination of filters.

4.2 A composition filters object in *Sina/st*

As mentioned in the previous section, the interface layer of a composition filters object is realized in *Sina/st* by the *interface part* of a class definition, and the inner object is realized by the *implementation part* of a class definition. This section describes the structure of these parts by using class *Stack* as an illustrative example. Class *Stack* provides operations like push, pop, top and size. An error is raised when an element is popped from an empty *Stack*, or when the top element is requested of an empty *Stack*.

In *Sina/st* an application can be started by using the **main** method. This is demonstrated in section 4.2.3.

4.2.1 The interface part

Example 4-1A shows the definition of the interface part in *Sina/st*. The name of the class, *Stack*, is placed between the keywords **class** and **interface**. We can recognize the components from figure 4-3: they are marked by the respective keywords.

The ordering of the keyword sections are as indicated. When the class has no components defined for a section, the keyword can be omitted. The minimum interface of a class is thus comprised of the keyword **class**, its name and the keywords **interface** and **end**.

Comment The section starting with the keyword **comment** defines the comment of the class interface. This kind of comment can only appear at this specific location. A comment starting with *//* can appear everywhere and extends to the end of the line.

Externals Externals, and also internals, are declared as a named instance of some class. Since the external objects lay outside the object, it is checked whether they exist and where they are located when the object is created. Class *Stack* has no externals.

Internals When an instance of the class is created, its internals are created. Since every internal is declared as an instance of some class, their internals, if there are any, are created, and so on.

The internal default of class *Stack* is declared as an instance of the class *SinaObject*. This class provides default operations, such as printing, testing and comparison.

EXAMPLE 4-1A

```
class Stack interface
  comment
    'This class represents a stack containing elements of arbitrary type.';
  externals
    // This class has no externals, but they could be declared as
    // anExt : ClassName;
  internals
    default : SinaObject;
  conditions
    stackNotEmpty;
  methods
    push(object : Any) returns nil;      // Stores the argument object.
    pop returns nil;                     // Removes the top element
    top returns Any;                     // Answers the top element
    size returns SmallInteger;           // Answers the number of elements
  inputfilters
    stackEmpty : Error = { stackNotEmpty => {pop, top}, true ~> {pop, top} };
    invoke : Dispatch = { true=>inner.*, true=>default.* };
  outputfilters
    send : Send = { true=>* };
end; // Stack interface
```

Conditions In the **conditions** section, the names of the conditions are specified. Conditions can be used in a filter to test whether to accept or to reject a message. The class *Stack* has only one condition, *stackNotEmpty*, which reflects whether the stack is empty or not.

Methods The **methods** section defines methods which are available to other objects. The parameters and the type of the value returned by these *interface* methods are specified. Apart from interface methods, a class can have *private* methods which can only be accessed by the class itself. Private methods are declared in the implementation part.

Input filters In the **inputfilters** section, an arbitrary number of filters are declared. A filter declaration defines the name of the filter, the filtertype and the filterinitializer.

The *filter initializer*, the part between the curly braces {}, specifies which messages under which conditions are accepted by the filter. The filter initializer is comprised of a number

of *filter elements*, each of which specifies a condition and a set of messages to be accepted. The filter elements are evaluated from left to right: if the message is not accepted by one filter element, for instance when the condition is invalid, then the message could still be accepted by a following filter element.

For example, the initializer of `Stack`'s first input filter, the error filter named `stackEmpty`, specifies that the filter accepts messages with the selector `pop` or `top` when the stack is not empty. This is expressed by the first filter element `stackNotEmpty=>{pop,top}`. It could be read as “if `stackNotEmpty` then accept a message with (pop or top)”.

The second filter element `true~>{pop,top}` could be interpreted as “if `true` then accept a message *not* with (pop or top)”, which is the same as “accept a message without pop and without top”. This filter element specifies that the filter further accepts messages with any selector *except* `pop` and `top`. As a result, the error filter rejects messages with selector `pop` or `top` when the stack is empty, causing it to generate an error, and accept messages otherwise.

Messages that have passed the error filter, are filtered by the following filter, the dispatch filter `invoke`. This filter accepts messages that are supported by the inner object (the filter element `true=>inner.*`) or that are supported by the internal default (`true=>default.*`). By definition, the messages supported by the inner object are the interface methods, which are `push`, `pop`, `top` and `size` in this case. When the dispatch filter accepts a message, it invokes the corresponding method of `inner`, or offers the message to the object `default`. This object filters the message and eventually invokes a method. Suppose that the `default` object also supports a method `size`, then, by the evaluation of filter elements from left to right, still the `size` method of `inner` is invoked.

From this we can conclude that an instance of class `Stack` supports the messages `push`, `pop`, `top`, `size` and all the messages which are defined by class `SinaObject`. The message `pop` or `top` sent to an empty stack generates an error.

Output filters

Like the former section, an arbitrary number of filters can be declared in the **outputfilters** section. The send filter `send` sends all messages (the filterelement `true=>*`) sent by a `Stack` instance to the receiver of the message.

4.2.2 The implementation part

The implementation part of a class in *Sina/st* is shown in example 4-1B. The name of the class is repeated between the keywords **class** and **implementation**. The implementation part has sections which respectively define a comment, the instance variables, the implementation of the conditions, an initial method and the implementations of private and interface methods. Again, when the class has no components defined for one of these sections, the corresponding keyword can be omitted.

Comment

A **comment** section in the implementation part can appear just after **implementation** and further in conditions and methods.

Instance variables

The instance variables of the class are defined in the **instvars** section. Like internals and externals, they are declared as a named instance of some class. The instance variables of an object are created after the internals have been created. The instance variable storage contains the elements of the stack, while the `SmallInteger stackPtr` points to the last used location in this Array.

Conditions

The implementation of the conditions are defined in the **conditions** section. A condition must return a boolean result. In the condition `stackNotEmpty`, the state of the stack is determined with the expression `stackPtr > 0`.

<i>Initial method</i>	The initial method can be used to initialize the instance. It is invoked at instance creation time, after the internals and instance variables have been created. The class Stack uses it to initialize the stackPtr .
<i>Methods</i>	In the methods section, the implementation of the interface methods are defined. Implementations of private methods can also be found here.
EXAMPLE 4-1B	<pre> class Stack implementation comment 'The instance variable "stackPtr" points to the last used location of the stack. The instance variable "storage" holds the elements of the stack'; instvars stackPtr : SmallInteger; storage : Array; conditions stackNotEmpty comment 'This state is true when the stack contains elements'; begin return stackPtr > 0; end; initial comment 'Start with an empty stack'; begin stackPtr := 0; end; methods push(object : Any) comment 'Pushes the argument object onto the stack'; begin stackPtr := stackPtr + 1; storage.at:put:(stackPtr, object) end; top comment 'Answers the top element of the stack'; temps object : Any; begin object := storage.at:(stackPtr); return object end; pop comment 'Removes the top element from the stack'; begin stackPtr := stackPtr - 1; end; size comment 'Answers the number of elements on the stack'; begin return stackPtr end; end; // Stack implementation </pre>
<i>Local variables</i>	Local variables of a method, and also condition, are defined after the keyword temps . The method top , for instance, uses a local variable named object to retrieve the top element of the stack.
<i>Statements</i>	The body of a method, surrounded by begin and end , can contain any number of statements and expressions. An assignment statement stores a value—an object—in a variable. <i>Sina/st</i> further defines statements like if-then-else , a while -loop and a for -loop. A value can be returned from a method, but also from a condition, by using the return statement. The condition stackNotEmpty and the methods top and size demonstrate this.
<i>Expressions</i>	There are two types of expressions in <i>Sina/st</i> : an operator expression, like stackPtr + 1 , and a message expression, like storage.at:(stackPtr) in which the message at: with argument stackPtr is sent to the object storage . In fact, an operator expression is a shorthand notation for a message expression. The compiler converts every operator expression to an equivalent message expression: stackPtr + 1 is replaced by stackPtr.plus(1) .

4.2.3 The main method

The main method can be used start an application. In general, the main method is very small. Example 4-2 shows this in which the class `Stack` from example 4-1 is applied and tested.

EXAMPLE 4-2

```
main
  comment 'Application of class Stack';
  externals
    Transcript : TextCollector;
  temps
    stack : Stack;
  begin
    stack.push('1st element');
    stack.push(2.0d0);
    stack.push('3rd element');
    while stack.size > 0
      begin
        Transcript.print(stack.top);
        Transcript.cr;
        stack.pop
      end;
    Transcript.flush
  end
```

The **main** method has the same structure as a method implementation. The only difference is that global objects that are referred to by main must be declared in the **externals** section. The external `Transcript` in example 4-2 is a global Smalltalk object which can be used to display messages on. The variable `stack` is local to main and has been declared in the **temps** section. The body shows that the stack is filled with three elements. Then, they are displayed on the `Transcript` and removed from the stack in the **while**-loop.

4.3 Forthcoming chapters

The rest of this part of the manual discusses the *Sina/st* language in more detail.

4.3.1 Syntax

Appendix A contains the complete syntax of the *Sina/st* language in the form of syntax diagrams. In following chapters, when we explain certain elements of the language, we present the syntax in a textual notation. The production for the conditional statement, for example, is denoted as

```
conditionalStatement ::=
  if expression
    then statements
  ( else statements )?
  end
```

A production consists of a nonterminal, called the left side of the production, a **::=** symbol, and a sequence of terminals, nonterminals and metasympols, called the right side of the production. Terminals are symbols which are shown in boldface font, like the semicolon **;** and the keywords **if**, **then**, **else** and **end**. For every nonterminal, shown in normal font, there is a production defined. Tokens on the right side of the production can be grouped by the metasympols **(** and **)**. A token or a token group followed by the metasympol **?** defines that the token, c.q. group is optional. The metasympol ***** defines a sequence of zero or more occurrences of the token or group. Finally, the metasympol **|** between token sequences specifies a choice between them.

4.3.2 Terminology

In the following chapters, we often refer to the term *first-class object* and the terms *owner*, *owning object* or *owning class*. They are defined here.

First-class object

According to [Wegner87], first-class objects can “be assigned to variables, be passed as parameters and be components of structures”. To put it in another way: a first-class object can be stored somewhere, while a non first-class object is only available during execution and cannot be stored. An active message, an objectmanager and a filter are examples of non first-class objects. Some non first-class objects can be accessed, like first-class objects, by sending a message to it. These kinds of objects include an active message and the object manager, because they are represented by the pseudo variables *message*, *^self* and *^server*. A filter on the other hand, cannot be accessed by sending a message to it.

Owner

The *owning class* or *owning object* of an entity (for instance, a filter or an internal) is the class, cq. the particular instance of the class, in which the entity was defined. By the term *owner* we mean the owning class or owning object, depending on the context. We can speak of the owner of an internal, a filter, a method, etc. For instance, the owner of the error filter *stackEmpty* in example 4-1A is the class *Stack*. In example 4-2, its owning object would be the object *stack*.

4.3.3 Structure of a *Sina/st* program

A *Sina/st* program consists of a single main method, or one or more class declarations, optionally followed by a main method. The syntax is thus:

```
program ::=
  main | classDeclaration ( classDeclaration ) * main?
```

4.3.4 Where to find what

Each of the following chapters addresses a specific topic of *Sina/st*. The overview below shows the subjects covered by each chapter.

Chapter	5	Basic language elements <i>literals</i> : characters, strings, numbers, nil, true, false; <i>variables</i> : local variables, method parameters, instance variables, internals, externals, class parameters, global externals; <i>pseudo variables</i> : inner, self, server, sender, message, ^self, ^server;
Chapter	6	Expressions <i>message expression</i> ; <i>operator expression</i> ;
Chapter	7	Control structures <i>conditional statement</i> ; <i>for statement</i> ; <i>while statement</i> ;
Chapter	8	Classes and instances <i>interface part</i> ; <i>implementation part</i> ; <i>instance creation</i> ; <i>object manager</i> : class <i>ObjectManager</i> ; <i>default behavior</i> : class <i>SinaObject</i> ;
Chapter	9	Methods <i>interface method</i> ; <i>private method</i> ; <i>initial method</i> ; <i>main method</i> ; <i>returning a value</i> ; <i>normal vs. early return</i> ; <i>statements</i> ;
Chapter	10	Conditions <i>reused condition</i> ; <i>local condition</i> ;
Chapter	11	Filters <i>input filters</i> ; <i>output filters</i> ; <i>reused filter</i> ; <i>local filter</i> ; <i>filtering a message</i> ; <i>signature of an object</i> ; <i>filtertypes</i> : dispatch filter; error filter, send filter, wait filter, meta filter;
Chapter	12	Message passing semantics <i>sending a message</i> ; <i>creating concurrency</i> ; <i>thread</i> ; <i>message reification</i> ; <i>message dereification</i> ; <i>active message</i> : pseudovariable <i>message</i> , class <i>ActiveMessage</i> ; <i>reified message</i> : class <i>SinaMessage</i> ;

5

Basic language elements

This chapter describes the basic language elements in *Sina/st*. The first section discusses the comment. Section 5.2 defines the literal constants, such as characters, strings, numbers, and the special literals `nil`, `true` and `false`. The following section describes variables and their declaration, while section 5.4 discusses the global externals. The last section defines the various pseudo variables.

5.1 Comment

Two forms of comment are allowed in a *Sina/st*: a one-line comment and a comment that begins with the keyword **comment**.

The one-line comment starts with two slash-characters (*//*) and extends to the rest of the line. It can contain any character except the newline character, of course.

The second form of comment consists of the keyword **comment** followed by a string literal and a closing semicolon character. The string literal, as we will see in section 5.2.2, can contain any character including a newline character, and therefore this type of comment can extend over several lines. This type of comment can only appear at fixed locations:

- the interface part of a class (refer to section 8.1.1);
- the implementation part of a class (section 8.1.2);
- the method implementation (sections 9.1-9.3);
- the main method implementation (section 9.4).
- the condition implementation (section 10.2);

The example below shows the use of both an interface comment and a one-line comment

```
class Stack interface
  comment 'You should describe here the function
          and behavior of class Stack';
  ...
end; // Stack interface
```

5.2 Literal constants

Sina/st provides three types of literal constants:

- characters;
- strings;
- numbers.

In addition, three special constants are defined: `nil`, `true` and `false`.

5.2.1 Characters

A character literal is made up of an initial `$`-character followed by one character which can be any character including a newline, a tab or a space character. The following literals are all examples of character literals:

```
$z
$$
$P
$3
$
$.
```

Every character literal is an instance of class `Character`.

5.2.2 Strings

A string literal is enclosed in single quote characters (`'`) and can contain any number of characters which can be any character including a newline, a tab or a space character. To include the single quote character itself, it must be preceded with a single quote character. The following four string literals illustrate this.

```
'To be or not to be?'           // a string with several characters
'A tab    and a newline
character can be included'      // a string containing a tab and a newline.
''                               // an empty string
''''                            // a string containing only one quote character
```

Every string literal is an instance of class `String`.

5.2.3 Numbers

There are two types of number literals: integer literals which do not have a decimal point embedded in them, and floating point literals which do. A negative number is represented, as usual, with a preceding minus sign. There must be no space between the minus sign and the literal, since the minus sign would otherwise be interpreted as the negation operator (see section 6.3, page 41).

```
1496.0           // positive floating point number 1496
-1496            // negative integer number -1496
- 1496           // negated positive integer number 1496
- -1496          // negated negative integer number -1496
```

Number literals can also be expressed in scientific notation by including an exponential part. The exponential part is composed of a preceding `"e"` or `"d"` and the exponent in decimal. The exponential part starting with an `"e"` can be used both with an integer and a floating point literal, whereas the exponential part starting with an `"d"` can be used only with a floating point literal. A floating point literal having an exponential part that starts with a `"d"` represents a double-precision number. A floating point literal without an exponential part or with one that starts with an `"e"` represents a single-precision number.

```

13e2                // integer number 1300
-13.0e2             // single-precision floating point number -1300
-13.0d2             // double-precision floating point number -1300

```

Integer literals are instances of class `SmallInteger`. Every floating point literal having an exponential part which starts with a “d” is an instance of class `Double`. A floating point literal without an exponential part or with one that starts with an “e” is an instance of class `Float`.

```

-1496               // a SmallInteger
1496.0              // a Float
13e2                // a SmallInteger
-13.0e2             // a Float
-13.0d2             // a Double
0.0                 // the Float 0.0
0.0e0               // another representation for the Float 0.0
0.0d0               // the Double 0.0

```

5.2.4 The constants nil, true and false

The object `nil` denotes an undefined object. The `nil` object is the sole instance of class `UndefinedObject`.

The boolean constants `true` and `false` are the sole instances of classes `True` and `False`, respectively, which are both subclasses of class `Boolean`.

5.3 Variables

Sina/st provides six types of variables:

- local variables of a method, a condition or main. They are declared in the **temps** section of the method, condition or main respectively;
- parameters of a method. They are declared in the header of the method;
- instance variables of an object. They are declared in the **instvars** section of the **implementation** part of a class;
- internal objects. They are declared in the **internals** section of the interface of a class. A class can use an internal to compose its behavior from;
- external objects. They are declared in the **externals** section of the interface of a class. Externals can be used to compose the behavior from and to share data among objects.

In *Sina/st version 3.00*, externals can only be global externals (see section 5.4) stored in the `SinaRepository`. Future versions will implement full scope rules which allow externals also to be internal variables, instance variables, or local variables of an encapsulating object;

- parameters of a class which are used to initialize an instance of a class. A class parameter behaves like an instance variable of a class. Class parameters are declared in the **interface** part of a class.

Variables must have been *declared* before they can be used (sections 5.3.1 and 5.3.2). Once a variable has been declared, it has an *initial value* –an object– which is assigned automatically or comes from the environment (5.3.3). The value of a variable can be changed by *assigning* a new object to it (5.3.4). The `nil` object can be assigned to a variable to indicate that its value is undefined.

*Variables are references
to objects*

In *Sina/st*, every variable is a *reference* to an object. As a consequence, method and class parameters are passed by reference too. If an object is not referred to by other objects, its memory is reclaimed automatically.

The consequence of variables being object references is that when one variable is assigned to another variable, they both refer to the same object.

```
main
  externals Transcript : TextCollector;
  temps john, mary : Person;
  begin
    john.setName('John');
    mary.setName('Mary');
    john := mary;
    john.setName('Johnny');
    Transcript.print:(mary.name);
    Transcript.cr;
  end;
```

In the example above, the variables `john` and `mary` initially refer to two different `Person` objects, one named 'John' and another named 'Mary', respectively. Once `mary` has been assigned to `john`, they both refer to the same object, the `Person` object with the name 'Mary'. If we change `john`'s name to 'Johnny', `mary`'s name will be changed at the same time. As a result, 'Johnny' is printed, when `mary`'s name is requested.

5.3.1 Declaration

A variable declaration defines the name and the type of each variable. Variables with the same type can be declared in the same declaration by separating their names by commas.

```
variableDeclaration ::=
  variableName ( , variableName )* : typeDescription
```

The name of a variable is an identifier that consists of a sequence of uppercase or lowercase letters (A-Z, a-z) and digits (0-9) which must start with a letter. The name(s) and the type are separated by a colon character. In the current version (*Sina/st version 3.00*), the colon *must* be preceded by a space, a tab or a newline character, otherwise the compiler issues a syntax error.

```
temps
  i, counter : SmallInteger;
  arr : Array(10);
```

In the **temps** section above, `i` and `counter` are both declared as variables of type `SmallInteger`. The variable `arr` is declared of type `Array(10)`, that is, an array with 10 elements of unspecified type. In these declarations, `SmallInteger` and `Array(10)` are so called *typedescriptions*.

5.3.2 Typedescription

A *typedescription* describes the type of the variable. It refers to a class and can have arguments. For some variables, it is also used to create their initial value (see below). There are four forms of *typedescriptions*:

```
typeDescription ::=
  Any
  | className ( ( expression ( , expression )* ) )?
  | className selector ( ( expression ( , expression )* ) )?
  | variableName
```

The type **Any**

The type **Any** can be used for variables whose type do not matter. An object of arbitrary type can be assigned to such a variable. The initial value of variables of type **Any** and which are automatically created (see below) is `nil`.

*Ordinary
typedescription*

The ordinary typedescription consists of the name of a class, optionally with argument expressions which are used to initialize the instance. Instances of Sina classes are constructed using this form, but it can also be used for the declaration of a Smalltalk instance. In the following example, the instance variable `bb` is declared as an instance of the Sina class `TypedBuffer`. It expects two arguments, the size of the buffer and the type of its elements.

instvars

```
bb : TypedBuffer(2 * n, SmallInteger);
```

`TypedBuffer(2 * n, SmallInteger)` is the typedescription of the variable `bb`. It has two argument expressions, `2 * n` and `SmallInteger`. Note that the argument `SmallInteger` is an object here, and not a typedescription. Therefore, it should have been declared as an external of type `Class`, otherwise the compiler flags it as an ‘Object not declared’-error.

*Typedescription
with constructor*

The third form of typedescription consists of the name of a Smalltalk class followed by a constructor and, optionally, argument expressions. The constructor is a Smalltalk selector and is sent to the class to construct an instance of that class. Note that this form is only allowed for Smalltalk classes, and that this class must have a class method¹ with the mentioned selector. It is responsibility of the programmer to assure that this class method does return a new instance of the class. Table C-2, page 117, describes the relationship between this form and Smalltalk. We show example declarations using this typedescription.

temps

```
today : Date today;
center : Point x:y:(260,435);
circle : Circle center:radius:(center, 50);
pi : Double pi;
```

The variable `today` is initialized with today’s date, `center` is a `Point` instance with initial `x` and `y` coordinates, and the variable `circle` is a `Circle` instance with a radius of 50 and the former `center` variable as center. Note that we can use previously defined variables in the typedescription argument expressions. The last declaration shows that the variable `pi` is initialized with the `Double` value of the constant π .

*Dynamic
typedescription*

The final form is the dynamic typedescription. It consists of the name of a variable, called the typevariable, which *must* have been declared as a variable of the type `Class`. This form can be used to declare variables of dynamic type. Such a dynamically typed variable is initialized with an instance of the class referred to by the typevariable. Because the parameters of a class cannot be specified in this form, only classes which have no parameters can be used safely as value of the typevariable. Classes which require parameters, such as the earlier mentioned `TypedBuffer` class, can be used, but their parameters are initialized with `nil`. Whether this will lead to (run-time) errors, depends on the robustness of that class.

```
newInstanceOfType(t : Class) returns t;
```

```
temps instance : t;
```

```
begin
```

```
    return instance
```

```
end;
```

```
exampleMethod
```

```
temps obj1, obj2, obj3 : Any;
```

```
begin
```

```
    obj1 := inner.newInstanceOfType(SmallInteger);
```

```
    obj2 := inner.newInstanceOfType(Array);
```

1. We are talking about Smalltalk classes here. A class method of a Smalltalk class will be executed by the class, whereas an instance method will be executed by an instance of that class. Some class methods will return a new instance of the class, while other are used for different purposes.

```

obj3 := inner.newInstanceOfType(TypedBuffer);
end;

```

The parameter *t* of the method `newInstanceOfType` above, is the typevariable for the local variable `instance`. This method returns an instance of type *t* that is passed as a parameter to the method. In the `exampleMethod`, `obj1` will be a `SmallInteger`, `obj2` will be an (empty) `Array`, and `obj3` will be a `TypedBuffer` instance with `nil` as size and `elementtype`.

5.3.3 Initial value

A variable has an initial value, that is, the variable initially refers to some object. Parameter variables are initialized with the actual argument objects. Except for externals, to the other types of variables a new instance is assigned. The new instance is constructed according to variable's typedescription. An external variable refers to an existing object — outside its owner— and is therefore not created. When their owner is created, it is verified whether the externals exist.

Table 5-1 shows for each type of variable whether an instance for it is created automatically and when this is done. For Smalltalk classes, an instance is constructed according to appendix C.1 (page 117).

TABLE 5-1

Initial value of variables

Kind of variable	Instance constructed automatically	When
temps	yes	at method invocation time
method parameters	no	passed as actual argument of a message
instvars	yes	at the instance creation time of the owning object
internals	yes	at the instance creation time of the owning object
externals	no	their existence are verified when the owning object is created
class parameters	no	passed as actual arguments of a variable declaration

5.3.4 Assignment

With the assignment statement, the value of a variable can be changed. It has the following syntax.

```

assignmentStatement ::=
    variableName := expression ;

```

All variables can be changed with an assignment, with one exception: it is not allowed to change the value of a method parameter². By assigning a new object to an external or an internal, the behavior of the owner could be modified. This is in general considered dangerous, but it is allowed. The compiler therefore issues a warning when this happens.

5.4 Global externals

A global external is an object which is stored in the `SinasytemDictionary Sina`. This is a collection of global objects which can be used as an external of a class or main method.

2. Due to the implementation in Smalltalk. Smalltalk does not allow method parameters to be changed either.

Smalltalk classes and objects are also contained in the SinaRepository. By declaring the Smalltalk object as an external with its class as type, the object can be accessed in a Sina class or in main.

In the following example, the Smalltalk classes Time and Double are used to display the current time and the value of the constant π on the Smalltalk system Transcript.

```
main
  externals
    Time, Double : Class;
    Transcript : TextCollector;
  begin
    Transcript.print:(Time.now);
    Transcript.cr;
    Transcript.print:(Double.pi);
    Transcript.cr;
    Transcript.flush
  end
```

*Creating a new
global external*

When in **main** a value is assigned to an external that did not exist before, it causes the creation of that external which is stored in the SinaRepository automatically. A value assigned to an existing external in main only changes the value of the global external.

By declaring a local variable —**temp**— in **main** of the same type as the non existing external and assigning this object to the external, the external is created and stored in the SinaSystemDictionary Sina.

The following example shows the creation of the external Hello of class HelloWorld which has been defined in example 3-1 (page 19).

```
main
  externals Hello : HelloWorld;      // Hello does not exist yet
  temps newGlobal : HelloWorld ;
  begin
    Hello := newGlobal;             // Hello will now be stored in SinaRepository
  end
```

After executing it, you can select the **repository** option from the source code view <operate> menu to see that the SinaRepository now includes the object Hello.

5.5 Pseudo variables

There are seven pseudo variables whose values change according to the execution context, and cannot be changed with an assignment. They are:

- sender;
- server;
- self;
- inner;
- message;
- ^self;
- ^server.

These pseudo variables refer to objects involved in the execution of a single message. When a message is sent, it is filtered first. Eventually a method is invoked (see chapter 12 for a more elaborate description of message execution). The aforementioned pseudo variables can be referred to inside a condition during the filtering of the message, and inside a method during its execution.

	<p>The first three pseudo variables, <code>self</code>, <code>server</code> and <code>sender</code>, are first-class objects, while <code>inner</code>, <code>message</code>, <code>^self</code> and <code>^server</code> are not. The latter four pseudo variables can only be accessed by sending a message to them, but they cannot be assigned to a variable.</p>
<p><code>sender</code> <code>server</code></p>	<p>The pseudo variable <code>sender</code> refers to the object which has sent the message. The pseudo variable <code>server</code> refers to the object to which the message was sent: it is the first object to receive the message. The input filters of this object can decide to delegate the message to another object (an internal or external) in which case the input filters of that object filter the message. Alternatively the input filters may decide to delegate the message to <code>inner</code>. This results in the invocation of a method.</p>
<code>self</code>	<p>During the filtering of a message, the pseudo variable <code>self</code> refers to the object that currently is filtering it. Once the message has invoked a method, that is, during the method's execution, <code>self</code> refers to the object which is executing the method. The pseudo variables <code>self</code> and <code>server</code> do not refer to the same object when the <code>server</code> has delegated the message to another object than itself. They do refer to the same object when the <code>server</code> (i.e. the first receiver) delegated the message to itself.</p>
<code>inner</code>	<p>The pseudo variable <code>inner</code> refers to the inner object (see section 4.1.1, page 24). Unlike <code>self</code>, <code>inner</code> refers to the object without the filters. When a message is sent to <code>self</code>, the message is filtered by the output filters and then by <code>self</code>'s input filters. A message sent to <code>inner</code> on the other hand, does not pass filters, but invokes a method of the object directly.</p>
<code>message</code>	<p>The pseudo variable <code>message</code> can also be referred to inside a condition and method. In the former case it refers to the message being filtered, while in the latter case, it refers to the message which lead to the invocation of the method. The pseudo variable <code>message</code> is an instance of class <code>ActiveMessage</code>. The attributes of a message (selector, arguments, sender, server and current receiver) can be retrieved by sending the appropriate message to <code>message</code> (see section 12.2, page 83).</p>
<p><code>^self</code> <code>^server</code></p>	<p>The final two pseudo variables, <code>^self</code> and <code>^server</code>, are object managers related to <code>self</code> and <code>server</code>, respectively. They are instances of class <code>ObjectManager</code> and can be referred to inside a condition or a method. An object manager controls the access to an object: it keeps track of how many messages have been received or sent and which methods are executing (see section 8.3, page 52).</p>

6

Expressions

An expression represents some value: when an expression is evaluated it returns an object. This object could be used to send a message to, but it could also be assigned to a variable, or passed as parameter to a method.

EXAMPLE 6-1

```
Tarzan.setName('Tarzan', 'King of the Jungle');
Jane.age < Tarzan.age
```

Example 6-1 shows the use of two expressions. The first expression is a so called *message expression*: the message `setName` with two arguments is sent to the receiver object `Tarzan`. Note that the two arguments are expressions on their own. The second expression is a so called *operator expression*: the two message expressions `Jane.age` and `Tarzan.age` are combined with the less-than operator (`<`).

An operator expression can be used by the programmer to denote the commonly used short-hand notations, such as arithmetic and logical functions. The *Sina/st* compiler, however, will convert every operator expressions to an equivalent message expression automatically. The operator expression of example 6-1 is identical to the message expression `(Jane.age).less(Tarzan.age)`.

The following section focuses on the expression in more detail. Section 6.2 will discuss the message expression, while section 6.3 will show how each operator expression is converted into an equivalent message expression.

6.1 Expression

An *expression* can be used in several situations, such as sending a message to an object, assignment of a value to a variable, or as an argument to a message. When an expression is evaluated it returns an object. An expression can be one of the following entities:

- an operand object, which is defined below;
- a message expression (section 6.2);
- an operator expression (section 6.3).

Operand object

An *operand object* denotes a single object. It can also serve as the receiver in a message expression, or as an operand in an operator expression. An operand object can be:

- a literal constant: a number literal (e.g. 3.14), a character literal (e.g. \$p) or a string literal (e.g. 'King of the Jungle');
- one of the special literals nil, true or false;
- one of the pseudo variables self, server, or sender;
- a variable: the name of a local variable, a method parameter, an instance variable, an internal, an external, or a class parameter.

In example 6-1, the two string literal arguments of the message setName are operand objects. Likewise, the variables Jane and Tarzan serve as receivers in both expressions.

Note that the pseudo variables inner and message, and the object managers ^self and ^server do not represent first class objects. This means that they cannot be passed as an argument of a message or assigned to a variable on their own. They can be used as the receiver in a message expression, however.

The syntax of the entities we have defined in this section is as follows.

```
expression ::=
    operandObject | messageExpression | operatorExpression
```

```
operandObject ::=
    variable | numberLiteral | characterLiteral | stringLiteral | nil | true | false
    | self | server | sender
```

6.2 Message expression

A *message expression* denotes a message that is sent to an object. It has three main components: a receiver, a method selector and zero or more arguments. In the message expression Tarzan.setName('Tarzan', 'King of the Jungle'), the receiver is Tarzan, the method selector is setName, and the two arguments are the string literals 'Tarzan' and 'King of the Jungle'.

Receiver

The receiver of a message expression can be one of the following:

- an operand object (section 6.1);
- one of the pseudo variables inner or message;
- an object manager: ^self or ^server;
- an expression enclosed by parentheses;
- a message expression.

Method selector

The method selector is separated from the receiver object by a message send dot. A method selector refers to the name of a method and consists of a sequence alphanumeric characters. Apart from the name of a method, a Smalltalk keyword selector is also allowed as method selector.

The Smalltalk keyword selector can be used to send a message to an instance of a Smalltalk class. A Smalltalk keyword selector contains a non-empty sequence of alphanumeric words each ending with a colon. The method selector at:put: in the expression anArray.at:put:(i, 0) is an example of such a keyword selector. The Sina/st compiler will check, wherever it can, if the Smalltalk class has defined a method with such a selector.

Note that the method of a Smalltalk class has one of the following three types of selectors:

- a unary selector, which consists of alphanumeric characters only, for instance sin. A unary selector expects no arguments;

- a binary selector, which consists of non-alphanumeric characters. A binary selector has only one argument. The selector == is an example;
- a keyword selector, such as the at:put: selector. A keyword selector expects as many arguments as there are colons in the selector.

In *Sina/st*, a Smalltalk method with a unary or keyword selector can be called by just using that selector as the method selector of a message. For a method with a binary selector you must use the method selector as defined in table C-3, page 119.

EXAMPLE 6-2

3.14.sin	// Smalltalk unary selector	3.14 sin
'Tar'.cat('zan')	// Smalltalk binary selector	'Tar' , 'zan'
anArray.at:put:(i, 0)	// Smalltalk keyword selector	anArray at: i put: 0

The previous example shows how in *Sina/st* the Smalltalk selectors can be used. Note that in Smalltalk the arguments of a keyword selector are mingled with it. In *Sina/st*, however, the message arguments are joined together after the selector.

Message arguments

The arguments of a message expression are enclosed by parentheses and separated by commas. If a message expression has no arguments, the enclosing parentheses are omitted. An argument is an expression, again.

We conclude this section by giving the syntax of the message expression.

```
messageExpression ::=
    receiver . selector ( ( expression ( ; expression ) * ) ) ?

receiver ::=
    variable | numberLiteral | characterLiteral | stringLiteral | nil | true
    | false | self | server | sender | inner | message | ^self
    | ^server | ( expression ) | messageExpression
```

6.3 Operator expression

An *operator expression*, such as 3+4*5, improves the readability of the source code and allows the programmer to write a short-hand notation instead of the more tedious, but equivalent message expression 3.plus(4.times(5)). *Sina/st*'s compiler will convert each operator expression simply to an equivalent message expression. It is important to note that only the selector of the equivalent message expression will be sent to the receiver object: in the example above, 3 receives the a message with selector plus and not +, and 4 will receive times and not *.

An *operator expression* is made up of an *operator* and one or more *operands* each of which is an expression again: an operand object, an operator expression, or a message expression.

Table 6-1 lists the available operators, their priorities and their equivalent message expressions. For clarity, the table includes the parentheses () and the message send dot (.), although they are strictly spoken no operators. A message send will always evaluated before any operator is applied. Parentheses can be used to enforce an evaluation order.

Highest priority
evaluated first. Equal
priorities evaluated from
left to right

An operator with the highest priority will be evaluated before an operator with a lower priority. Operators with equal priority will be evaluated from left to right if and only if the operator is left-associative. A left-associative operator is indicated by a bullet • in the column "asso-ciates left" of table 6-1. A cross × in the same column indicates that the operator cannot be combined with operators having the same priority. For instance, Jane.age = Tarzan.age = 50 makes no sense and will be flagged by the compiler as a syntax error. By using parentheses, we could write (Jane.age = Tarzan.age) = false, though.

TABLE 6-1

Operators with their priority and equivalent message expression

description	operator	priority	associates left	sample expression	equivalent message expression
force order	()	>7	×	(a / b).floor	
message send	.	>7	●	a.atput(b,c)	
logical negation	not	7	●	not a	a.not
numeric negation	-		●	- a	a.negated
product	*	6	●	a * b	a.times(b)
exact division	/		●	a / b	a.divide(b)
integer division	div		●	a div b	a.div(b)
integer remainder	mod		●	a mod b	a.mod(b)
addition	+	5	●	a + b	a.plus(b)
subtraction	-		●	a - b	a.minus(b)
comparison	<	4	×	a < b	a.less(b)
„	<=		×	a <= b	a.upto(b)
„	>		×	a > b	a.greater(b)
„	>=		×	a >= b	a.atleast(b)
value equality	=	3	×	a = b	a.equal(b)
value inequality	!=		×	a != b	a.unequal(b)
identity equality	==		×	a == b	a.same(b)
identity inequality	!=		×	a != b	a.different(b)
logical disjunction	and	2	●	a and b	a.and(b)
logical conjunction	or	1	●	a or b	a.or(b)

The *Sina/st* compiler transforms an operator expression into a message expression as listed in the column labeled "equivalent message expression" of table 6-1. The priorities of the operators will be taken into account.

EXAMPLE 6-3

Jane.age < Tarzan.age	Jane.age.less(Tarzan.age)
(b*b - 4*a*c).sqrt	((b.times(b)).minus(4.times(a).times(c))).sqrt
-x+ y.cos + x*y/4	(x.negated).add(y.cos).add(x.times(y).divide(4))
a>=0 and a<=10 or a=-1	(a.atleast(0).and(a.upto(10))).or(a.equal(-1))
10 - -3	10.minus(-3.negated)

The example above shows how some operator expressions are converted into their equivalent message expression. The last expression demonstrates the negation and subtraction of the negative number literal -3. Note that the minus sign - serves three roles: as the dyadic subtraction operator, as the monadic negation operator and as the sign of a number literal.

Notes

Some remarks on the operations of table 6-1 will be made here.

- The integer division operator div is defined as division with truncation toward negative infinity. Thus, $9 \text{ div } 4 = 2$, $-9 \text{ div } 4 = -3$ and $-0.9 \text{ div } 0.4 = -3$. The integer remainder operator mod yields the corresponding remainder (modulo): $9 \text{ mod } 4 = 1$, $-9 \text{ mod } 4 = 3$, $9 \text{ mod } -4 = -3$, $0.9 \text{ mod } 0.4 = 0.1$.

Use the method selector `quo:` for integer division with truncation toward zero. Thus, `-9.quo:(4) = -2` and `-0.9.quo:(0.4) = -2`. The method `rem:` gives the corresponding remainder: `9.rem:(4) = 1`, `-9.rem:(4) = -1`. `0.9.rem:(0.4) = 0.1`.

- The operators `=` and `!=`, and the corresponding method selectors `equal` and `unequal`, are used to test whether the values of two objects are equal. The operators `==` and `!=`, and the respective method selectors `same` and `different`, test whether the two operands (the receiver and the argument object) are the same object. Thus, `4 = 4.0` yields `true`, whereas `4 == 4.0` yields `false`, since the `SmallInteger` object `4` and the `Float` object `4.0` represent the same values, but are two different objects.
- Case differences are ignored when two strings are compared by the operators `=`, `!=`, `<`, `<=`, `>`, or `>=`, and the respective method selectors `equal`, `unequal`, `less`, `upto`, `greater` or `atleast`. This is due to implementation of the comparison methods of the `Smalltalk` class `String`.

Thus, `'A' < 'a'` and `'A' > 'a'` both evaluate to `false`, even though the ASCII value of the character `$A` is less than the ASCII value of `$a`. The expression `'A' = 'a'` would evaluate to `true`, and therefore `'A' != 'a'` yields `false`. The comparison methods do, however, consider the length of the strings: `'a' < 'ab'` and `'a' > 'ab'` yield `true` and `false`, respectively.

To compare strings considering case differences, you should use the method `trueCompare:` which returns `-1`, `0` or `1` if the receiver is less than, equal to, or greater than the argument. Thus, `'A'.trueCompare:'a'` yields `-1`.

We conclude this chapter by giving the syntax of an operator expression.

```
operatorExpression ::=
    monadicOperator operand
| operand dyadicOperator operand
```

```
operand ::=
    operandObject | operatorExpression | messageExpression
```

```
monadicOperator ::=
    not | -
```

```
dyadicOperator ::=
    or | and | = | != | == | != | < | <= | => | > | - | + | * | /
    | div | mod
```


7

Control structures

Sina/st defines three control structures, one conditional selection statement and two iterative constructs:

- the conditional statement;
- the for statement;
- the while statement.

7.1 The conditional statement

The conditional statement can be used to choose between two alternatives. Its syntax is as follows.

```
conditionalStatement ::=
  if expression
    then statements
    ( else statements )?
  end
```

Evaluation of the *expression* must yield an instance of class `Boolean`, that is, either `true` or `false`. If it yields `true`, the statements in the **then**-part will be executed. The **else**-part is optional, but when present, the statements in it will be executed when the *expression* yielded `false`. The following example determines the maximum and minimum of two variables `a` and `b`.

EXAMPLE 7-1

```
if a < b
then min := a; max := b
else min := b; max := a
end
```

7.2 The for statement

The for statement is used for iteration over numbers. It has the following syntax.

```

forStatement ::=
  for loopvar := startExpression to endExpression ( by stepExpression )?
  begin
    statements
  end

```

The variable *loopvar* must have been declared before it can be used, for instance as a local variable or as an instance variable. Inside the loop, it is not allowed to change the value of *loopvar*, but you can assign a value to it outside the loop. The value of *loopvar* after the loop has finished is not defined. Before the iteration of the loop, the *startExpression*, *endExpression* and *stepExpression* will be evaluated respectively to determine the start value, end value and step value of the loop. The **by** *stepExpression* can be omitted, in which case a step value of 1 is used. The loopvariable *loopvar* will be initialized with the start value. The statements in the body will be executed as long as the value of *loopvar* does not exceed the end value. After each iteration, the step value will be added to the value of *loopvar*. For positive step values this means that the loop iterates while the value of the *loopvar* is less than or equal to the end value, and for negative step values while the value of the *loopvar* is greater than or equal to the end value.

In the following example, *f*'s values are calculated from -2 to 2 in increments of 1/8 and plotted on the screen.

EXAMPLE 7-2

```

for x := -2.0 to 2.0 by 0.125
begin
  screen.plot(x, f.value(x) )
end

```

7.3 The while statement

The while statement is used for iteration while some condition is met. It has the following form.

```

whileStatement ::=
  while expression
  begin
    statements
  end

```

Evaluation of the *expression* must yield an instance of class Boolean, that is, either true or false. The statements in the body will be executed repeatedly until the evaluation of the *expression* yields false.

The following example could be part of a window controller which tracks the mouse until a button is released.

EXAMPLE 7-3

```

while mouse.buttonPressed
begin
  mouse.track
end

```

8

Classes and instances

In this chapter we will describe the structure of a class (section 8.1) and how an instance of a such a class is created (section 8.2). Section 8.3 describes the object manager which controls such an instance.

8.1 Class declaration

A class declaration consists of two parts, the interface part and the implementation part. The interface part can be preceded by class switches which can be used to control properties of a class. The class switches will be described after the interface and implementation part.

```
classDeclaration ::=
  ( classSwitch ) * interfacePart implementationPart
```

It is not allowed to define a class with the same name as an existing Smalltalk class, although an existing Sina class can be redefined.

8.1.1 The interface part

The interface part of a class defines

- its name after the keyword **class**;
- its parameters between the class name and the keyword **interface**. The parameters are surrounded by parentheses which must be omitted if the class has no parameters;
- an interface comment in the section with the keyword **comment**;
- the externals in the section with the keyword **externals**;
- the internals in the section with the keyword **internals**;
- the conditions in the section with the keyword **conditions**;
- interface methods in the section with the keyword **methods**;
- the set of input filters in the section with the keyword **inputfilters**;
- the set of output filters in the section with the keyword **outputfilters**;

The general form of the interface part of a class is shown below. Some sections or parts of them can be omitted. The minimal interface of a class is comprised of the keyword **class**, its name and the keywords **interface** and **end**.

```
interfacePart ::=
class className classParameters? interface
  ( comment string ; )?
  ( externals ( variableDeclaration ; )* )?
  ( internals ( variableDeclaration ; )* )?
  ( conditions ( conditionDeclaration ; )* )?
  ( methods ( interfaceMethodDeclaration ; )* )?
  ( inputfilters ( filterDeclaration ; )* )?
  ( outputfilters ( filterDeclaration ; )* )?
end ;

classParameters ::=
  ( variableDeclaration ( ; variableDeclaration )* )
```

8.1.2 The implementation part

The implementation part of a class defines

- its name after the keyword **class**. The name must be the same as in the interface part;
- an implementation comment in the section with the keyword **comment**;
- the instance variables in the section with the keyword **instvars**;
- the condition implementations in the section with the keyword **conditions**;
- the implementation of the initial method in the section with the keyword **initial**;
- the method implementations in the section with the keyword **methods**;

The syntax of the implementation part is shown below. Some sections or parts of them can be omitted. The minimal implementation of a class is comprised of the keyword **class**, its name and the keywords **implementation** and **end**.

```
implementationPart ::=
class className implementation
  ( comment string ; )?
  ( instvars ( variableDeclaration ; )* )?
  ( conditions ( conditionImplementation ; )* )?
  ( initial ( methodBody ; )* )?
  ( methods ( methodImplementation ; )* )?
end ;
```

8.1.3 Class switches

A class switch can be used to control properties of a class, such as default behavior and mutual exclusion. A switch will apply to a single class. Currently supported class switches are:

```
#Category 'A category';
#DefaultObject;
#NoDefaultObject;
#DefSyncFilter;
#NoDefSyncFilter;
#DefBehaviorFilter;
#NoDefBehaviorFilter;
```

*Default behavior and
synchronization*

The keywords of the switches are not case-sensitive: `#defaultobject`; `#DEFAULTOBJECT`; and `#DefaultObject` all denote the same switch. The switches will be described below.

When a class has no switches defined, the class will provide default behavior and mutual exclusion. The compiler will realize this by inserting

- an internal default of type `SinaObject`;
- an input wait filter `defSync` which provides mutual exclusion for every instance of the class;
- the conditions `free` and `recursive` which are used in the `defSync` filter;
- an input dispatch filter `defBehavior` which provides default behavior for every instance of the class;

`#Category 'A category';`

This switch controls the category in the Smalltalk system dictionary in which the compiled class will be stored. When this switch is not specified, the class will be stored in the category defined in the user interface window (refer to section 3.1, page 15).

`#DefaultObject;`
`#NoDefaultObject;`

These switches define whether or not an internal called `default` of type `SinaObject` will be inserted by the compiler automatically. Class `SinaObject` implements default behavior for a class, such as printing, copying, etc. (see appendix B.1, page 113). The definition of the internal is:

```
internals
  default : SinaObject;
```

The object default will be inserted

- if the switch `#DefaultObject`; has been specified, or
- if the switch `#NoDefaultObject`; has *not* been specified.

The object default will be omitted

- if the switch `#NoDefaultObject`; has been specified, or
- if the class already has an external, internal or instance variable of an arbitrary type called `default`.

When the internal default is inserted, it will be inserted as the *first* internal of the class. Table 8-1 illustrates where the internal default will be inserted.

TABLE 8-1

Location of the internal default.

// no switch specified	<code>#DefaultObject</code> ;	<code>#NoDefaultObject</code> ;
<pre>class X interface externals ... internals default : SinaObject; a : A; b : B; ... conditions ... methods ... inputfilters ... outputfilters ... end;</pre>	<pre>class X interface externals ... internals default : SinaObject; a : A; b : B; ... conditions ... methods ... inputfilters ... outputfilters ... end;</pre>	<pre>class X interface externals ... internals a : A; b : B; ... conditions ... methods ... inputfilters ... outputfilters ... end;</pre>

`#DefSyncFilter;`
`#NoDefSyncFilter;`

These switches determine whether or not the input wait filter `defSync` will be inserted. This filter provides mutual exclusion for each instance of the class. The input wait filter `defSync` has the following definition:

```
defSync : Wait = { {free, recursive}=>inner.*, true~>inner.*};
```

The input wait filter defSync will be inserted

- if the switch #DefSyncFilter; has been specified, or
- if the switch #NoDefSyncFilter; has *not* been specified.

The input wait filter defSync will be omitted

- if the switch #NoDefaultObject; was specified, or
- if the switch #NoDefSyncFilter; was specified, or
- if the class already has an input filter called defSync.

When the input wait filter defSync is inserted, it will be inserted before the first input filter that is not a wait filter. When no input filters are defined, or all the input filters are wait filters, the defSync filter will be placed after the last after input filter.

When this filter is inserted, the conditions free and recursive will also be inserted unless they are already defined. Their implementations are defined as:

```

conditions
  free      begin return ^self.active = 0      end;
  recursive begin return message.isRecursive end;
```

The defSync filter provides mutual exclusion. Messages with a selector defined by the class self, i.e. which are in the inner signature, will be blocked by the filter when there is already a local method active. Other message, those in the signature of internals or externals will never be blocked by this filter. The location of the defSync filter are illustrated in table 8-2.

TABLE 8-2

Location of the input wait filter defSync and the conditions free and recursive.

no switch specified	#DefSyncFilter;	#NoDefSyncFilter;
<pre> class X interface externals ... internals ... conditions ... free; recursive; methods ... inputfilters f1 : Wait = {...}; f2 : Wait = {...}; defSync : Wait = { ...} ; f3 : NonWait = {...}; f4 : NonWait = {...}; f5 : Wait = {...}; ... outputfilters ... end;</pre>	<pre> class X interface externals ... internals ... conditions ... free; recursive; methods ... inputfilters f1 : Wait = {...}; f2 : Wait = {...}; defSync : Wait = { ...} ; f3 : NonWait = {...}; f4 : NonWait = {...}; f5 : Wait = {...}; ... outputfilters ... end;</pre>	<pre> class X interface externals ... internals ... conditions ... methods ... inputfilters f1 : Wait = {...}; f2 : Wait = {...}; f3 : NonWait = {...}; f4 : NonWait = {...}; f5 : Wait = {...}; ... outputfilters ... end;</pre>

#DefBehaviorFilter;
#NoDefBehaviorFilter;

These switches define whether or not the input dispatch filter defBehavior will be inserted. This filter provides the behavior of the default object. The input dispatch filter defBehavior has the following definition:

```
defBehavior : Dispatch = { default.* };
```

The input dispatch filter `defBehavior` will be inserted

- if the switch `#DefBehaviorFilter`; has been specified, or
- if the switch `#NoDefBehaviorFilter`; has *not* been specified.

The input dispatch filter `defBehavior` will be omitted

- if the switch `#NoDefaultObject`; was specified, or
- if the switch `#NoDefBehaviorFilter`; was specified, or
- if the class already has an input filter called `defBehavior`.

When the input dispatch filter `defBehavior` is inserted, it will be inserted before the first input filter which is not a wait filter. When no input filters are defined, or all the input filters are wait filters, the `defBehavior` filter will be placed after the last after input filter. The location of the filter is shown in table 8-3.

TABLE 8-3

Location of the input dispatch filter `defBehavior`.

// no switch specified	#DefBehaviorFilter;	#NoDefBehaviorFilter;
<pre> class X interface externals ... internals ... conditions ... methods ... inputfilters f1 : Wait = {...}; f2 : Wait = {...}; defBehavior : Dispatch = {...} ; f3 : NonWait = {...}; f4 : NonWait = {...}; ... outputfilters ... end;</pre>	<pre> class X interface externals ... internals ... conditions ... methods ... inputfilters f1 : Wait = {...}; f2 : Wait = {...}; defBehavior : Dispatch = {...} ; f3 : NonWait = {...}; f4 : NonWait = {...}; ... outputfilters ... end;</pre>	<pre> class X interface externals ... internals ... conditions ... methods ... inputfilters f1 : Wait = {...}; f2 : Wait = {...}; f3 : NonWait = {...}; f4 : NonWait = {...}; ... outputfilters ... end;</pre>

8.2 An instance of a class

As described in section 5.3.3, internals, instance variables and local variables are variables which will be initialized with a newly created instance of the class specified in the variable declaration.

When an instance of a class is created, or *constructed* as it will be called here, its internals and instance variables will be constructed. This means that every internal and instance variable will be initialized with an instance of the class specified in the variable declaration.

The same applies to the local variables (**temps**) of a method or main. When the method or main is invoked, its local variables will be constructed: each variable will be initialized with an instance of the class specified in the variable declaration.

An external, a class or method parameter will not be constructed, since, in case of an external, the object already exists, and in case of a parameter, the object will be passed by the caller.

When an instance `x` of a Sina class, say `X`, is constructed, the following actions will be carried out:

- The class parameters of `X`, if any, will be initialized with the values of the arguments of `x`'s declaration at the time of its construction;

- The existence of the class' externals will be verified. If there are externals that do not exist, these will be reported on the object's window and the creation of the instance is aborted with the error 'Can't create instance Y::x', where x is the instance of class X declared as some internal, instance variable or local variable in class Y;
- The internals will be constructed. This can, recursively, lead to the construction of instances of other classes. It is therefore not allowed to declare an internal of the same class from which it is part of;
- The instance variables will be constructed. Again, this can lead to the construction of instances of other classes. It is neither allowed to declare an instance variable of the same class from which it is part of;
- And finally, when class X has defined one, its initial method will be executed.

8.3 The object manager

Every instance has associated with it a so-called object manager. As the name implies, an object manager controls an object. It keeps track of the number of methods that have been invoked and finished, and the number of messages that have been blocked by a wait filter.

An object manager can be referred to by the pseudo variable `^self` and `^server`. The former refers to the to the object manager of `self` while the latter refers to the object manager of `server`. An object manager is an instance of the class `ObjectManager` which provides the following methods (see also appendix B.4, page 116):

TABLE 8-4

Methods of an object manager

<code>active</code> returns <code>SmallInteger</code> ;	Answers the number of active (unfinished) method invocations in the object. Note that after an early return, a method still is active (executing). The method will finish only when its last statement has been executed.
<code>activeForMethod(selector : String)</code> returns <code>SmallInteger</code> ;	Answers the number of active (unfinished) invocations of the method given by the argument.
<code>blockedReq</code> returns <code>SmallInteger</code> ;	Answers the number of messages that have been blocked by an <i>input</i> wait filter.
<code>blockedInv</code> returns <code>SmallInteger</code> ;	Answers the number of messages that have been blocked by an <i>output</i> wait filter.

9

Methods

In *Sina/st*, there are three kinds of methods:

- interface method;
- private method;
- initial method.

The interface method is declared at the interface part of the class and can be called by any client object. A private method can only be invoked by its owner. Finally, the initial method is executed automatically at instance creation time, right after the internals and instance variables have been created. Sections 9.1 to 9.3 discuss the three method types in more detail.

main

Apart from these methods, which are bound to a class, *Sina/st* also defines the main method which can be used to start an application. The main method is the subject of section 9.4.

Return and early return
Creating concurrency

A method returns a value —an object— when it executes a `return` statement or the equivalent object manager message `^self.reply()` (see section 9.5). For normal methods, the return is the last statement that is executed by the method. In *Sina/st*, however, it is also possible that the method continues with the execution of statements after the it has returned a value. In this case, the method is a so called *early return* method. The statements following the return are executed concurrently with the method that invoked the early return method. With an early return method it is thus possible to create concurrency. Section 9.6 describes when a method is a normal or an early return method, while section 9.7 shows the difference between the invocation of a normal method and an early return method.

Finally, section 9.8 describes the statements which are allowed in a method body.

9.1

*The declaration of
an interface method*

Interface method

An interface method must have been declared at the interface part of a class. They are declared in the section of the interface part which starts with the keyword `methods`. An interface method declaration defines the name of the method, the names and types of the parameters and the type of the object returned from the method, its so-called *returntype*. If

the returntype is specified as nil, it means that the method always returns the nil object, i.e. that it does not return a relevant object.

The syntax of an interface method declaration is shown below.

```
methodDeclarations ::=
  ( methods ( interfaceMethodDeclaration ; )* )?

interfaceMethodDeclaration ::=
  methodName methodName? returns returnType

methodName ::=
  identifier

methodParameters ::=
  ( variableDeclaration ( ; variableDeclaration )* )

returnType ::=
  nil | typeDescription
```

The example below shows the interface of class Person which contains the declaration of three interface methods, setName, birthDay and age.

EXAMPLE 9-1 A

```
class Person interface
  comment ";
  externals
  internals
  conditions
  methods
    setName(firstname, surname : String) returns nil;
    birthDay returns Date;
    age returns SmallInteger
    // ... other interface methods
  inputfilters
    disp : Dispatch = {inner.*};
  outputfilters
end; // Person interface
```

The first method, setName, does not return a value and therefore its returntype is nil. The two other methods, birthDay and age, return an instance of class Date and SmallInteger, respectively. The setName method has two parameters of the same type: firstname and surname both are Strings. The two other two methods, on the other hand, do not have any parameters.

*Making an
interface method
available for a client*

For an interface method to be able to be invoked by a client object, it is necessary that a Dispatch filter is declared in the group of input filters which contains a filter element (see section 11.3.2, page 71) that refers to the method. The target of such a filter element must be inner and its selector must match with the name of the method: this is the case if the selector of the filter element either is a wildcard or is same as the name of the method.

In example 9-1 A, the input Dispatch filter disp includes such a filter element. The filters below exhibit some other combinations of filter elements for the interface methods of example 9-1. The f1 filter, for example, uses substitution on a message with selector dateOfBirth in order to invoke the method birthDay. Thus, if an object sends a message dateOfBirth to a Person, the birthDay method will be invoked and the date of birth of the Person will be returned.

```
f1 : Dispatch = { [* .dateOfBirth]inner.birthDay, nameUnset => inner.setName };
f2 : Dispatch = {..., c2 => inner.*, ...};
```

*The implementation
of an interface method*

The section of the implementation part of a class which starts with the keyword `methods`, embodies the implementations of interface methods. The ordering of the methods is not important: the methods can be in a different order as they are declared at the interface part and private methods can be mixed in between as well. The syntax of the implementation of an interface method is as follows:

```
interfaceMethodImplementation ::=
  methodName methodParameters?
  ( comment string ; )?
  ( temps ( variableDeclaration ; ) * )?
  begin statements end
```

The implementation of an interface method duplicates the names and types of the parameters from the declaration at the interface, and defines further the comment, the temporary variables and the body of the method. Note that the return type is not repeated in the implementation. The example 9-1 B shows the implementation part of class `Person` with implementations of the aforementioned interface methods.

*How to call
an interface method*

An interface method can be called by sending a message, with a selector that is the same as the name of the method, to an object that supports the method. The `setName` method of class `Person`, for instance, could be called in the following ways:

- `p.setName('John', 'Smith');`
In this case, the sending object sends the message to an object `p` which should be an instance of class `Person` (or even an instance that eventually delegates the message to an instance of class `Person`);
- `self.setName('John', 'Smith');`
Here, the sending object sends a message to itself. By sending it to `self`, the message passes through the input filters of the object (after it passed through its output filters, to be precisely, see section 11.2, page 70);
- `inner.setName('John', 'Smith');`
In this case, the sending object sends the message to `inner`, which calls the method directly, that is, without passing through the output filters or input filters.

The implementation of the method `age` shows also a message sent to `inner`. The selector of this method, `ageOn`, does not refer to an interface method though, but to a private method. The private method is described in the coming section.

EXAMPLE 9-1 B

```
class Person implementation
  comment ";
  instvars
    birthDay : Date;
    firstName, lastName : String;
    // ... other instvars
  conditions
  initial
  methods
    setName(firstName, surname : String)
      comment 'Define new firstName and lastName';
      temps    // ... no temps
      begin
        firstName := firstName;
        lastName := surname;
      end; // setName

    birthDay
      comment 'Answer my date of birth.';
      begin
        return birthDay
      end; // birthDay
```

```

age
    comment 'Answer my current age';
    temps today : Date today;
begin
    return inner.ageOn(today)
end; // age

// ... other methods
end; // Person implementation

```

9.2

*Joint declaration
and implementation*

Private method

A private method implements some behavior that only available to the owning object. Therefore, a private method is not declared in the interface part of a class, but its declaration and implementation are combined and defined in the implementation part. The implementation of a private method is similar to the implementation of an interface method. The only difference, though, is that the returntype of the private method *is* defined in the header:

```

privateMethodImplementation ::=
    methodName methodParameters? returns returnType
    ( comment string ; )?
    ( temps ( variableDeclaration ; ) * )?
    begin statements end

```

In the following example, the private method `ageOn` returns a `SmallInteger` which denotes the Person's age on the day that is passed as an argument to the method. Note that in this specific implementation, the Smalltalk class `Date` is used and also the Smalltalk selectors `subtractDate:` and `fromDays:`.

EXAMPLE 9-2

```

class Person implementation
    // ... same as in example 9-1 B
    methods
        // ... other methods

        ageOn(day : Date) returns SmallInteger
        comment 'Answer the age of this person on the argument date "day" ';
        instvars
            dateDiff, date0 : Date;
        begin
            dateDiff := Date.fromDays:(day.subtractDate:(birthDay));
            date0 := Date.fromDays:(0);
            return dateDiff.year - date0.year
        end; // ageOn
end; // Person implementation

```

*How to call
a private method*

A private method can be invoked by the owning object only, by sending a message to the pseudo variable `inner`. In this way, the message does not pass the input filters, nor the output filters, but the method is invoked directly. In the interface method `age` (see example 9-1 B) the private method `ageOn` is called with `today`'s date as an argument.

9.3

Initial method

The initial method is optional and will be executed, if it has been defined, just after the instance has been created, including its internals and instance variables. This method allows, for instance, to initialize internals and instance variables with specific values. The initial method is specified in the implementation part of a class, between the conditions and methods sections.

The returntype of the initial method is not specified, but is, by definition nil: the initial method does not return a value. Neither does it have parameters. Like the private and interface methods, the implementation of the initial method defines a comment, temporary variables, both optionally, and the method body:

```
initialMethodImplementation ::=
  initial
  ( comment string ; )?
  ( temps ( variableDeclaration ; ) * )?
  begin statements end ;
```

The initial method for class Person is shown in the following example. The name of every new Person instance will be initialized with 'Nomen Nescio'.

EXAMPLE 9-3

```
class Person implementation
  comment ";
  instvars
    // ... from example 9-1 B
  conditions
  initial
    comment 'Define firstName and lastName';
    temps
    begin
      firstName := 'Nomen';
      lastName := 'Nescio';
    end; // initial
  methods
    // ... from examples 9-1 and 9-2
end; // Person implementation
```

9.4 Main method

The **main** method can be used start an application. In general, the **main** method is very small, but you can use it to test newly defined classes. We have used the **main** method already several times. The syntax form of the main method is as follows:

```
main ::=
  main
  ( comment string ; )?
  ( externals ( variableDeclaration ; ) * )?
  ( temps ( variableDeclaration ; ) * )?
  begin statements end
```

The **externals** and **temps** sections are used to declare externals and local variables, respectively. In **main**, an external refers to a global external stored in the SinaSystemDictionary Sina. The **main** method can also be used to create a new external (see section 5.4, page 36). The body can contain any number of statements.

The **main** method returns nil. It is possible to perform an early return in **main**. The **main** method can be executed by selecting the **run** option from the <operate> menu in the source code area (see section 3.1.2, page 17).

9.5 Returning a value from a method

A value —an object— can be returned from a method in two ways:

- by execution of the return statement, or
- by sending the message reply to the object manager ^self.

^self.reply message

The `^self.reply(...)` message is an ordinary message expression (see section section 6.2, page 40). The argument of the reply message is an expression that specifies the value to be returned by the method.

Return statement

A return statement consists of the keyword **return** and is optionally followed by an expression. This expression denotes the value to be returned by the method. The expression following **return** can only be omitted if the return value of the method is specified as nil.

The return statement is actually equivalent to the `^self.reply(...)` message. Table 9-1 shows the equivalence between the return statement and the objectmanager message `reply`. The *Sina/st* compiler translates a return statement into its corresponding reply message.

TABLE 9-1

Equivalence of return statement and object manager message reply

return	<code>^self.reply(nil)</code>
return nil	<code>^self.reply(nil)</code>
return expression	<code>^self.reply(expression)</code>

The interface methods `birthDay`, `age` (example 9-1 B), and the private method `ageOn` (example 9-2) all show the application of the return statement to yield a value from the method. By using the objectmanager message `reply`, the method `age`, for instance, could also be written as follows.

EXAMPLE 9-4

```
age
  comment 'Answer my current age';
  temps today : Date today;
  begin
    ^self.reply( inner.ageOn(today) )
  end;
```

*Implicit return if no
return expression
in method body*

If a method body does not include a return expression (i.e. a return statement or a `^self.reply` message), the return statement **return** is implicitly the last statement in the method body. Note that this is only possible for methods whose return value is specified as nil: the compiler will flag an error if the return value of the method is not nil and its body does not contain a return expression.

The interface method `setName` (example 9-1 B) and the initial method (example 9-3) of class `Person` are illustrations of methods with an implicit return statement.

Early return

Normally, the method finishes its execution when the method has returned a value. However, if other statements follow the return in the body of the method, the method will continue to execute these statements. This kind of method is called an *early return method* and is subject of the next section.

9.6 Normal versus early return

In this section we define more precisely if a method is a normal method or an early return method (definition 9-2). It is based on the notion called *control flow*. Further, we define requirements which must be obeyed by every control flow (definitions 9-3 to 9-5). This section ends by giving examples of early and normal methods.

DEFINITION 9-1

Control flow in a method

A control flow in a method (or flow for short) is one possible execution of statements of the method body. The flow starts with the first statement of the method body, then the next, etcetera.

There are several flows of control possible if a method body contains a conditional or a loop statement. In a conditional statement, depending on the value of the condition, the

flow can either follow the then or else branch. The statements in the body of a while or for loop can be executed zero or more times.

For the following method body we have indicated the possible flows of control. One flow, labeled f_1 , starts by evaluating the condition $c1$ and, if $c1$ was true, continues in the then branch by executing the statement $s1$ and returning object x (denoted in the flow by $\wedge x$). Further, there is an infinite number of other flows, all indicated by $f_2(i)$. They all start by evaluating $c1$ to false (denoted by $\sim c1$) and then continue with the execution of the while loop. The statement $s2$ in the body of the while loop will be executed as long as the condition $c2$ evaluates to true (denoted by $(c2, s2)^i$). The flow ends by returning the object y after $c2$ becomes false (shown as $\sim c2, \wedge y$).

```

begin
  if  $c1$ 
  then  $s1$ ; return  $x$ 
  else
    while  $c2$  begin  $s2$  end
  end;
  return  $y$ 
end;
```

$f_1 = c1, s1, \wedge x$

$f_2(i) = \sim c1, (c2, s2)^i, \sim c2, \wedge y$ with $i=0,1,2,\dots$

Now that we have defined the control flow in a method, we define when a method is a normal or an early return method.

DEFINITION 9-2

Early return method and normal return method

A method is an early return method if there is a flow in which the return statement is followed by at least one other statement.

Consequently, a method is a normal method if the last statement in every flow is a return statement.

Examples of normal and early return methods are given at the end of this section. In the following section the difference between the invocation of a normal and an early return method is explained. But first we define two rules that must be valid for every control flow in a method.

DEFINITION 9-3

Control flow requirements

The following two requirements must hold for every control flow of a method:

1. *In every flow there must be a return.*
2. *Only one return is allowed in a flow.*

The first requirement is needed because the method must always return a value. If this not the case, the compiler generates the error ‘No value returned in one flow’.

The second requirement is needed because a method can return a value only once. The compiler produces the error ‘Return after early return is not allowed’ if this requirement is disregarded.

The first three of the following method bodies do not follow the first requirement, while the last breaches the second. The offending elements are highlighted.

begin s1; s2; s3; end;	begin if c then s1; return x; end; s2; end;	begin while c1 begin s1; if c2 then return x; end; end;	begin s1; return x; s2; return y; s3; end;
$\bar{f} = s1, s2, s3$	$\bar{f}_1 = c, s1, \wedge x$ $\bar{f}_2 = \sim c, s2$	s3; end; $\bar{f}_1(i) = (c1, s1, \sim c2)^i, \sim c1, s3$ $\bar{f}_2(i) = (c1, s1, \sim c2)^i, c1, s1, c2, \wedge x$ with $i=0,1,2,\dots$	$\bar{f} = s1, \wedge x, s2, \wedge y, s3$

We introduce two further rules, one for the conditional statement and another for the two loop statements. They are not really necessary since they are already covered by the two control flow requirements, but they ‘make life easier’.

DEFINITION 9-4

Control flow requirement for the conditional statement

If in a conditional statement either the then-branch or the else-branch has an early return, there must be a return in the other branch too.

The statements following the conditional statement are executed after the execution of either branch. These statements are not allowed to have another return, because of the early return in one branch. The flow via the other branch thus would have no return, thereby breaching the first control flow requirement. The compiler generates the error ‘Early return in one flow, no return in the other’ in case this rule is not satisfied.

DEFINITION 9-5

Control flow requirement for the while and the for statement

An early return inside the body of a loop statement (a while statement or a for statement) is not allowed, though a normal return is.

This rule is needed because the body of the loop can be executed several times, and therefore the (early) return could be executed more than once too. A breach of this rule causes the compiler to generate the error ‘Early return not allowed inside loop’.

The following two method bodies display violations of these rules.

// Early return in one flow, // no return in the other begin if c then s1; return x; s2 else s3; end; s4; end;	// Early return not allowed // inside loop begin while c begin s1; return x; s2; end; end;
$\bar{f}_1 = c, s1, \wedge x, s2, s4$ $\bar{f}_2 = \sim c, s3, s4$	$\bar{f}(i) = (c, s1, \wedge x, s2)^i, \sim c$ For instance if $i=2$: $\bar{f}(2) = c, s1, \wedge x, s2, \sim c, s1, \wedge x, s2, \sim c$

Finally, this section presents examples of normal and early return methods. Example 9-5 A shows bodies of normal methods and their associated flows of control. Example 9-5 B shows the method bodies of some early return methods together with their flows.

EXAMPLE 9-5 A

Normal method bodies and their control flows.

begin s1; s2; s3; return x; end ; <hr/> $f_1 = s1, s2, s3, \wedge x$ <hr/>	begin if c then s1; else s2; end ; s3; return z; end ; <hr/> $f_1 = c, s1, s3, \wedge z$ $f_2 = \sim c, s2, s3, \wedge z$ <hr/>	begin if c then s1; return x; else s2; return y; end ; end ; <hr/> $f_1 = c, s1, \wedge x$ $f_2 = \sim c, s2, \wedge y$ <hr/>
begin if c1 then s1; return x; end ; if c2 then s3; return y; end ; s5; return z; end ; <hr/> $f_1 = c1, s1, \wedge x$ $f_2 = \sim c1, c2, s3, \wedge y$ $f_3 = \sim c1, \sim c2, s5, \wedge z$ <hr/>	begin if c1 then s1; return x; else s3; if c2 then s5; return y; else s7; return z; end ; end ; end ; <hr/> $f_1 = c1, s1, \wedge x$ $f_2 = \sim c1, s3, c2, s5, \wedge y$ $f_3 = \sim c1, s3, \sim c2, s7, \wedge z$ <hr/>	begin while c begin s1; s2; end ; s5; return z; end ; <hr/> $f(i) = (c, s1, s2)^i, \sim c, s5, \wedge z$ with $i=0,1,2,\dots$ <hr/>
begin while c begin s1; return x; end ; s2; return y; end ; <hr/> $f_1 = c, s1, \wedge x$ $f_2 = \sim c, s2, \wedge y$ <hr/>	begin while c1 begin s1; if c2 then s2; return x; else s3; return y; end ; end ; s4; return z; end ; <hr/> $f_1 = c1, s1, c2, s2, \wedge x$ $f_2 = c1, s1, \sim c2, s3, \wedge y$ $f_3 = \sim c1, s4, \wedge z$ <hr/>	begin while c1 begin s1; if c2 then s2; return y; end ; s3; end ; s4; return z; end ; <hr/> $f_1(i) = (c1, s1, \sim c2, s3)^i, \sim c1, s4, \wedge z$ with $i=0,1,2,\dots$ $f_2(i) = (c1, s1, \sim c2, s3)^{i-1}, c1, s1, c2, s2, \wedge y$ with $i=1,2,3,\dots$ <hr/>

EXAMPLE 9-5 B

Early return method bodies and their control flows.

<pre> begin s1; return x; s2; s3; end; </pre>	<pre> begin if c then s1; return x; else s3; return y; end; s5; s6 end; </pre>	<pre> begin if c then s1; return x; else s3; return y; s4 end; s5; s6 end; </pre>
$f_1 = s1, ^x, s2, s3$	$f_1 = c, s1, ^x, s5, s6$	$f_1 = c, s1, ^x, s5, s6$
$f_2 = \sim c, s3, ^y, s5, s6$	$f_2 = \sim c, s3, ^y, s5, s6$	$f_2 = \sim c, s3, ^y, s4, s5, s6$
<pre> begin if c1 then s1; return x; else s2; if c2 then s3; return y; else s4; return z; end; s5; end; </pre>	<pre> begin if c1 then s1; return x; else s2; if c2 then s3; return y; else s4; return z; end; end; s5; end; </pre>	<pre> begin while c begin s1; s2; end; s3; return x; s4; s5 end; </pre>
$f_1 = c1, s1, ^x$	$f_1 = c1, s1, ^x, s5$	$f(1) = (c, s1, s2)^1, \sim c, s3, ^x, s4,$
$f_2 = \sim c1, s2, c2, s3, ^y, s5$	$f_2 = \sim c1, s2, c2, s3, ^y, s5$	$s5$
$f_3 = \sim c1, s2, \sim c2, s4, ^z, s5$	$f_3 = \sim c1, s2, \sim c2, s4, ^z, s5$	with $i=0,1,2,\dots$

9.7 Method invocation

This section describes (1) when a method is invoked, (2) what happens upon and during the invocation of a method, and (3) the difference between the invocation of a normal and an early return method.

When does a method invocation start?

A method is invoked in one of the following cases:

- if a Dispatch filter accepts a message which was passing the input filters and dispatches it to inner. This is possible for interface methods only;
- if a message is sent to the pseudo variable inner. In this way, both interface methods and private methods can be invoked;
- at instance creation time. The initial method is invoked in this way and in this way only.

What happens upon method invocation?

Upon invocation of a method, the local variables (**temps**) of the method are created first: every local variable is initialized with a instance of the class which was specified by the declaration of the variable. Note that the parameters of a method —the arguments of the message— have been passed by reference because all variables are object references in *Sina/st*. What happens next depends on whether the invoked method is a normal or an early return method.

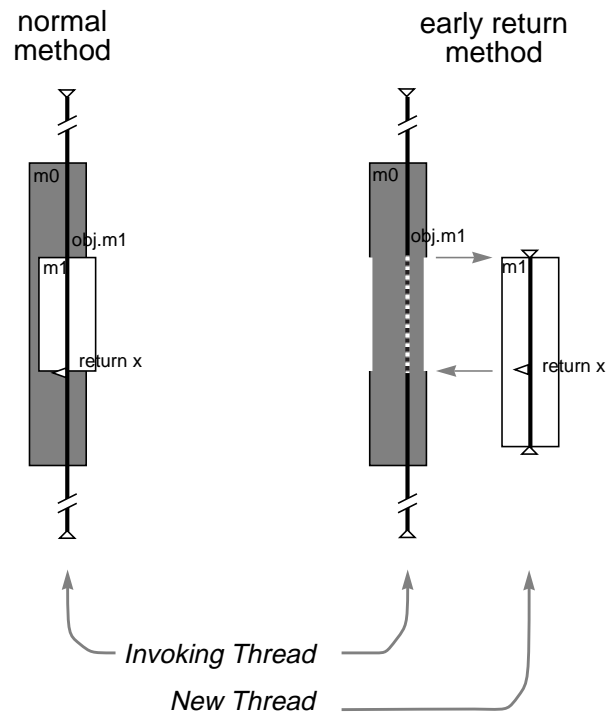
Figure 9-1 depicts those differences. An early return method creates a new thread¹, while a normal method does not. The start and the end of a thread are represented by an downward and an upward pointing triangle, respectively. The thread itself appears as a continuous or

1. A thread is a sequence of messages which can execute concurrently with other threads. See section 12.3, page 83.

dashed line, and a method as a rectangle. The figure below² shows two possible invocations of method m1 as the result of the so-called *invoking message* (obj.m1) which has been sent in the so-called *invoking method* m0.

FIGURE 9-1

Invocation of a normal method and an early return method.



Invocation of a normal method

If m1 is a normal method, the statements in the body of the method will be executed in the same thread in which the invoking message was sent. The statements in the body are executed one after another until a return is executed (**return x**). Then, the invoking method (m0) continues with the statement which follows the invoking message (obj.m1).

Invocation of an early return method

For an early return method (shown right in figure 9-1) a new thread is created and the statements of the method body are executed in this thread. The invoking method and its thread are blocked until the invoked method returns an object (**return x**). The invoking thread then resumes its execution: the remaining statements of the invoking method are executed concurrently with the statements following the return in the invoked method. The thread in this method ends when its last statement has been executed.

Using an early return method, we can create a new, concurrent thread. Note that a new thread is created every time an early return method is invoked.

Pseudo variables during the invocation

During the invocation of the method, the pseudo variables refer to the following objects:

- **message**: the invoking message. It is an instance of class `ActiveMessage`;
- **inner**: the inner part of the executing object, i.e. without the encapsulating interface part, without the filters;
- **self**: the executing object;
- **server**: the object that originally received the invoking message;
- **sender**: the object which has sent the invoking message.

2. This figure shows two diagrams similar to the diagrams used in chapter 12. In the diagrams used here, we have omitted the passing of the message through the filters, but we do show the invoking method.

9.8 Statements in a method body

The statements in the method body are separated by semicolons. It is possible to have an empty statement, and thus also to have an empty method body. A statement can further be an assignment (section 5.3.4, page 36), an expression (chapter 6), one of the control structures (chapter 7) or a return statement (section 9.5). The syntax is as follows:

```
statements ::=
    statement
| statements ; statement

statement ::=
    assignmentStatement
| messageExpression
| operatorExpression
| conditionalStatement
| forStatement
| whileStatement
| returnStatement
| // i.e. empty
```

10

Conditions

A condition provides information about the current state of an object. It is a special kind of method that takes no parameters and returns a boolean result. Apart from conditions defined by the class self, so called local conditions, it is possible to reuse a condition from another class.

A condition is declared at the interface part of a class (section 10.1), while the implementation of a local condition is defined in the implementation part (section 10.2). Note that the implementation of a reused condition is defined by the class it has been reused from. Section 10.3 describes the use of conditions.

10.1 Condition declaration

There are two forms of condition declaration, a reused condition and a local condition declaration. The reused condition defines the name of the condition and the object it is reused from. This object must be an external or an internal. The local condition declaration defines only its name, while its implementation is defined in the implementation part. It is not allowed that a condition and a method have the same name. The syntax of a condition declaration is shown below.

```

conditionDeclarations ::=
  ( conditions ( conditionDeclaration ; ) * ) ?

conditionDeclaration ::=
  localConditionDeclaration | reusedConditionDeclaration

localConditionDeclaration ::=
  conditionName

reusedConditionDeclaration ::=
  objectName . conditionName

conditionName ::=
  identifier

objectName ::=
  identifier

```

We show the declaration of two local conditions in the example below. Class `BoundedBuffer` allows you to store and retrieve objects using the `put` and `get` method, respectively. The condition `notFull` represent the state in which the buffer is not full yet. The condition `notEmpty` describes the state in which the buffer contains at least one element. An object can be stored in the buffer if the buffer is not full, while an object can be retrieved from it if the buffer is not empty. The conditions are used in the wait filter `bufferSync` to specify the synchronization of the buffer. Section 11.6.4 will discuss the wait filter in more detail.

EXAMPLE 10-1 A

```
class BoundedBuffer(limit : SmallInteger) interface
  conditions
    notFull;
    notEmpty;
  methods
    put(object : Any) returns nil;
    get returns Any;
  inputfilters
    bufferSync : Wait = {notFull => put, notEmpty => get};
    dispatching : Dispatch = {inner.*};
end; // BoundedBuffer interface
```

10.2 Condition implementation

The implementation of conditions are defined in the **conditions** section of the implementation part of a class. The ordering of the conditions is not important: the conditions can be in a different order as they are declared at the interface part. The condition implementation has a similar structure as a method implementation. A comment and local variables can be defined, while the body can contain any statement also allowed in the body of a method. The syntax of the condition implementation as follows:

```
conditionImplementations ::=
: ( conditions ( conditionImplementation ; ) * ) ?

conditionImplementation ::=
: conditionName
  ( comment string ; ) ?
  ( temps ( objectDeclaration ; ) * ) ?
begin statements end
```

As an example of condition implementations, we continue with the implementation of class `BoundedBuffer`, shown below. The buffer stores its elements in the instance variable `store`. The instance variables `putpos` and `getpos` hold the position in the array `store` where to store (method `put`) and retrieve (method `get`) the next object, respectively. Initially, they will both be at the first position, i.e. 1. The buffer is empty if `putpos` and `getpos` are the same. The buffer is full if the `putpos` is about to overtake the `getpos`, that is, if $(\text{limit} + \text{getpos} - \text{putpos}) \bmod \text{limit} = 1$. Note that the capacity (the maximum number of objects) of the buffer is $\text{limit} - 1$: the last free position, in this particular implementation, is needed to be able to recognize the full state.

The implementation of the `notFull` and `notEmpty` conditions are straightforward: they are the logical negation of the formerly derived full and empty constraints.

EXAMPLE 10-1 B

```
class BoundedBuffer implementation
  instvars store : Array(limit); // index ranges from 1 to limit
  putpos, getpos : SmallInteger;
  conditions
    notFull
  begin
    return not((limit + getpos - putpos) mod limit = 1)
```

```

        end;
    notEmpty
    begin
        return not(putpos = getpos)
    end;
initial
    begin putpos := 1; getpos := 1; end;
methods
    put(object : Any)
    begin
        store.at:put:(putpos, object);
        putpos := putpos mod limit + 1
    end;
    get
    temps object : Any;
    begin
        object := store.at:(getpos);
        getpos := getpos mod limit + 1;
        return object
    end;
end; // BoundedBuffer implementation

```

10.3

Condition invocation

Using a condition

A condition is mainly invoked during the filtering of a message: a filter uses a conditions to decide to accept or reject the message. A condition, however, can also be invoked by its owner by sending a message to inner. This can be used for creating a condition which is a logical combination of other conditions and in a method to provide state information of the object. It is however not possible for objects to call a condition by sending a message to its owner.

The notFull and notEmpty conditions of example 10-1 could be combined to provide a condition called partial which reflects the state if the BoundedBuffer instance is neither full nor empty. Its implementation can be defined as:

```

partial
    begin return inner.notFull and inner.notEmpty end;

```

The second use of invoking a condition is, for example, in a method printState of class BoundedBuffer which displays whether the buffer is full or not. This method could be defined as:

```

printState
    begin
        if inner.notFull
            then self.print('The buffer is not full')
            else self.print('The buffer is full')
        end
    end;

```

Pseudo variable message

If a condition has been invoked during the filtering of a message, the pseudo variable message refers to this message. It can be used to examine the attributes of the filtered message. For instance, the condition ifYouAreAPostman (example 11-4, page 76) tests whether the sender of a message is a postman, i.e. an instance of class Postman. It could have been defined as

```

conditions
    ifYouAreAPostman
        begin return message.sender.class == Postman end;

```

If a condition has been invoked by sending a message to inner, the pseudo variable `message` refers to this message.

Conditions must be side effect free: this means that the condition must not invoke local methods since this can lead to the release of blocked messages. The current compiler cannot check whether a condition is side effect free. It is therefore the responsibility of the programmer to provide such conditions.

Finally a warning. Do not send a message to `self`, `server` or `sender`, in the condition body, as this can or, in case of a message to `self`, certainly leads to a deadlock situation. This is understandable because the condition is evaluated while the message is filtered by the filtergroup. Only one message will be evaluated by a filtergroup at a time. Other arriving messages are blocked until the filtered message leaves the filtergroup. If a condition sends the aforementioned message, it waits forever for its reply.

11

Filters

This chapter discusses the various filter types and the filtering mechanism. First it introduces some terminology and general concepts.

11.1

Introduction

Every class specifies a group of *input filters* and a group of *output filters*. A message sent *to* an object must pass the input filters, and a message sent *by* an object must pass its output filters. Every filter in the group of filters *filter a message* that passes it. This means that the filter determines whether it accepts or rejects the message, and then takes an accept or reject action, respectively. A filter accepts a message if it is accepted by one of the *filterelements* specified in the *filterinitializer*. The message is rejected if no *filterelement* can accept the message. The *filterinitializer* contains a number of *filterelements*. A *filterelement* consists of a condition and a *messagepattern*: it accepts a message if the condition is true and the message matches the *messagepattern*.

The accept, c.q. reject action of a filter is determined by the *filterhandler* of the filter. The *filterhandlers* implemented in *Sina/st version 3.00* include Dispatch, Error, Send, Wait and Meta. A filter with such a handler is called a dispatch filter, an error filter, a send filter, a wait filter or a meta filter, respectively.

EXAMPLE 11-1

class Stack interface

```
...
inputfilters
  stackEmpty : Error = { stackNotEmpty => pop, true ~> pop};
  invoke : Dispatch = { inner.pop, inner.push};
  ...
end; // Stack interface
```

The example above shows a group of two input filters as part of the interface of class Stack. The filter called `stackEmpty` is an error filter —its handler is `Error`— and its initializer contains two *filterelements*: `stackNotEmpty => pop` and `true ~> pop`. If the stack is not empty, this filter will accept a message with selector `pop`, since the first *filterelement* accepts such a message: the condition `stackNotEmpty` evaluates to `true` and the selector of the message matches the *messagepattern* `pop`. The second *filterelement* `true ~> pop` is an exclusion *filterelement*. Such kind of *filterelement* will accept a message if the condition evaluates to `true` and the message does *not* match the message pattern. The exclusion

element true ~> pop thus accepts messages that do not have a selector pop. The stackEmpty error filter thus rejects a message pop if the stack is not empty and accepts other messages. The accept action of an error filter causes the message to be offered to the following filter, while the reject action raises an error.

The next filter is called invoke. This is a dispatch filter —its handler is Dispatch— which accepts a message with a selector pop (accepted by its first filterelement) or push (accepted by its second filterelement). The dispatch filter accept action dispatches the message to inner which causes the respective methods pop and push to be invoked. A message that is rejected by a dispatch filter is passed on to the next filter. If there are no further filters in the set, such as in this case, and a message has fallen through the last filter, an error will be raised.

11.2 Input and output filters

Every object has a group of *input filters* and a group of *output filters*. This has been depicted in figure 11-1. A group of filters acts like a layered collection of sieves: a message is sieved by each filter in that group in turn, until a filter filters it out. If a message has passed all the filters in the group, —i.e. no filter has filtered it out— the object cannot handle the message and it will raise an error.

Which messages pass the input filters?

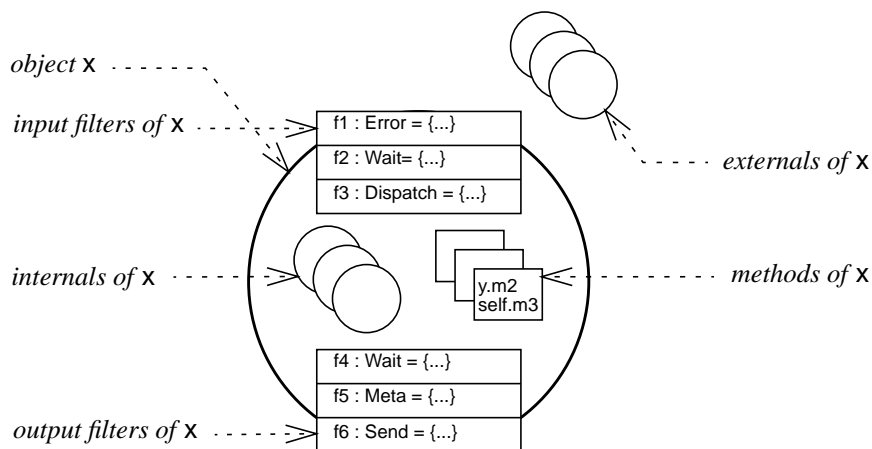
The group of input filters determines how a message that is *sent to* the object containing them, is handled by that object. Every message that is sent to an object passes the group of input filters of that object. For instance, in figure 11-1, a message $x.m(a_1, \dots, a_n)$ is sent to object x first passes the input filters of x . Suppose the input dispatch filter f_3 of x accepts the message and delegates it to an object z which is an internal or external of x : the message then passes through the input filters of z .

Which messages pass the output filters?

The group of output filters determines how a message that is *sent by* the object to another object is handled. Only a message sent to an object that lays outside the boundary of the sending object will pass the output filters of the sending object. That is, if an object sends a message to one of its externals, to self, server or sender, the message will pass the output filters. A message will not pass through the output filters if it is sent to the pseudovalue message, to an object manager $\wedge self$ or $\wedge server$, to an internal or an instance variable of the object, or to a local variable of a method, but enters the input filters of the receiver of the message directly. A message sent to inner does not pass through the output filters either, but since inner is the object without filters, a method is invoked immediately.

FIGURE 11-1

An object with input and output filters



In figure 11-1, in a method of the object x message $y.m2$ is sent. If y is an external of x , the message passes the output filters of x , otherwise y is an internal or instance variable of x , or a local variable of the method and therefore the message enters the input filters of y directly.

The group of input and output filters are declared in the class interface part, in the section starting with the keywords **inputfilters** and **outputfilters**, respectively.

```
inputfilters ::=
    inputfilters ( filterDeclaration ; ) *

outputfilters ::=
    outputfilters ( filterDeclaration ; ) *
```

11.3 Filterdeclaration

There are two forms of filter declarations: the declaration of a local filter, and the declaration of a reused filter. The former defines the filter handler and the filterinitializer self. The second form declares a filter that is reused from an external or internal object. This means that the filter handler and initializer is determined by the class of the external or internal object.

11.3.1 Reused filter

It is possible to reuse a filter from an other class with the reused filter declaration:

```
filterDeclaration ::=
    objectName . filterName
```

The *objectName* must be the name of an internal or external object. The *filterName* must be a name of either a reused filter declaration or a local filter declaration of the object to which the *objectName* refers.

The handler and the initializer are as that of the reused filter declaration. Objects (matching and substitution targets) and conditions that are referred to in the initializer are bound to those (in the scope) of the reusing object, i.e. *not* to those of the object from which the filter declaration is reused.

*A reused filter is bound
at compile-time*

In the current compiler version (version 3.00) only compile-time binding to the filter declaration is used. This means that if its handler or initializer changes (in class it is reused from) after an object that reused this filter has been compiled, this object is not updated with the new definition of the filter.

11.3.2 Local filter

With the local filter declaration, the class can specifies its name, handler and initializer:

```
filterDeclaration ::=
    filterName : filterHandler = filterInitializer
```

The *filtername* defines the name of the filter. The *filterhandler* defines the handler of the filter which must be one of Dispatch, Error, Wait, Send or Meta. The *filterinitializer* defines how a message is filtered by the filter.

Filterinitializer

The *filterinitializer* is comprised of at least one *filterelement*. Each *filterelement* determines if it accepts or rejects a message that passes through the filter. If a message is accepted by the *filterelement*, the message is accepted by the filter and its accept action will be carried out. If a message is rejected by a *filterelement*, the next *filterelement* (in declaration order, from left to right) will be consulted to see if that *filterelement* can accept the message. If no *filterelement* can accept the message, the message is rejected by the filter. In this case, its reject action will be carried out.

```
filterInitializer ::=
    { filterElement ( , filterElement ) * }
```

Filterelement

A *filterelement* determines if it accepts or rejects a message that passes through the filter. There are two forms: an *inclusionelement* and an *exclusionelement*.

The *inclusionelement* accepts a message if one of the conditions in the *conditionset* evaluates to true and the message matches one of the patterns in the *messagepatternset*. The *exclusionelement*, on the other hand, accepts a message if one of the conditions in the *conditionset* evaluates to true and the message does *not* match *any* of the patterns in the *messagepatternset*.

EXAMPLE 11-2

```
mySync : Wait = {m1, {c1, c2}=>b.*, true~>{b.*, m2}};
```

This example shows a wait filter which initializer has two *inclusionElements* and one *exclusionElement*. This filter would accept a message with selector m1 always, a message with a selector in the signature of b (see page 76) if either c1 or c2 is true, and a message with a selector that is not m2 and is not in the signature of b.

```
filterElement ::=
    inclusionElement | exclusionElement

inclusionElement ::=
    messagePattern | conditionSet => messagePatternSet

exclusionElement ::=
    conditionSet ~> messagePatternSet
```

Condition

A condition referred to in a filterinitializer is has to be declared in the interface of the class. It can either be a reused condition or a condition implemented by the class itself. The condition **true** is also possible which evaluates always to true.

```
conditionSet ::=
    condition | { condition ( , condition ) * }

condition ::=
    true | conditionName
```

Messagepattern

A *messagepattern* is used to match the selector or the target or both of the message. The *matchingpattern* and *signaturepattern* are the two forms possible.

In the *matchingpattern* [a.b]c.d, the message must match with the *matchingpart* a.b, and if it does, the *substitutionpart* c.d will be returned as the result of the match. With the *signaturepattern* c.d, the message matches if the selector of the message is c and is included in the signature of d.

```
messagePatternSet ::=
    messagePattern
    | { messagePattern ( , messagePattern ) * }

messagePattern ::=
    matchingPattern
    | signaturePattern

matchingPattern ::=
    [ matchingPart ] substitutionPart

matchingPart ::=
    messageSelector
    | objectName . messageSelector
    | * . messageSelector
    | self . messageSelector
```

```

substitutionPart ::=
  signaturePattern

signaturePattern ::=
  messageSelector
| objectName . messageSelector
| * . messageSelector
| inner . messageSelector
| self . messageSelector

messageSelector ::=
  selector | *

```

11.3.3 Rewrite rules for filterelements

Some parts of the filterelements can be omitted—for instance, the *conditionset* in an *inclusionelement*—while others have multiple parts, like the *conditionset*. By rewriting them using the following rules, one can derive equivalent *filterinitializer* which contains *inclusionelements* that have a single *condition* and a single *messagepattern*, and *exclusionelements* that have a single *condition* and either a single or a set of *messagepatterns*.

The rewrite rules for filterelements are listed in the following table. The first rule shows that the default target is the wildcard. A missing condition in a filterelement implies the true condition (rule 2). Rules 3 to 6 show that (set of) messagepattern(s) is distributed over the elements of a set of conditions. In the last rule, a single condition is distributed over the elements of a set of messagepatterns.

TABLE 11-1

Rewrite rules for filterelements

	Filterelement	Filterelement after rewrite
1	sel	*.sel
2	mp	true=>mp
3	{c ₁ , c ₂ , ..., c _n }=>mp	c ₁ =>mp, ..., c _n =>mp
4	{c ₁ , c ₂ , ..., c _n }=>{mp ₁ , ..., mp _p }	c ₁ =>{mp ₁ , ..., mp _p }, ..., c _n =>{mp ₁ , ..., mp _p }
5	{c ₁ , c ₂ , ..., c _n }~>mp	c ₁ ~>mp..., c _n ~>mp
6	{c ₁ , c ₂ , ..., c _n }~>{mp ₁ , ..., mp _p }	c ₁ ~>{mp ₁ , ..., mp _p }, ..., c _n ~>{mp ₁ , ..., mp _p }
7	c=>{mp ₁ , mp ₂ , ..., mp _p }	c=>mp ₁ , c=>mp ₂ , ..., c=>mp _p

Note that there exists no rule that distributes a set of conditions over a set of messagepatterns. The filterelement {c₁, c₂, ..., c_n}=>{mp₁, ..., mp_p} is therefore *not* equivalent to {c₁, c₂, ..., c_n}=>mp₁, ..., {c₁, c₂, ..., c_n}=>mp_p. Neither is the filterelement {c₁, c₂, ..., c_n}~>{mp₁, ..., mp_p} equivalent to {c₁, c₂, ..., c_n}~>mp₁, ..., {c₁, c₂, ..., c_n}~>mp_p.

EXAMPLE 11-3

The following four filterdeclarations are equivalent to each other by using the rules 1, 4 and 7, respectively.

```

defSync1 : Wait = { {free, recursive}=>{m3, inner.*}, true~>inner.*};
defSync2 : Wait = { {free, recursive}=>{*m3, inner.*}, true~>inner.*};
defSync3 : Wait = { free=>{*m3, inner.*}, recursive=>{*m3, inner.*}, true~>inner.*};
defSync4 : Wait = { free=>*m3, free=>inner.*, recursive=>*m3, recursive=>inner.*,
                    true~>inner.*};

```

But they are not equivalent to

```

defSync5 : Wait = { {free, recursive}=>*m3, {free, recursive}=>inner.*, true~>inner.*};

```

11.4 Filtering a message

This section describes the filtering of a message in detail. The filtering process of a message is made up of the steps shown below. The filterelement accept test of steps 4 to 8 have been combined in table 11-2 for an inclusionelement and in table 11-3 for an exclusionelement. The signature of an object which is referred to in step 5b is defined in section 11.5.

1. *filtering through the group of filters.*

The message is offered to each filter in the group in turn, starting with the first, *to filter* it. If the filter did not *sift it out*, it will be offered to the next filter. If the last filter in the group has not filtered it out, an error will be raised: 'Input filters didn't filter out message *M*' or 'Output filters didn't filter out message *M*' if the message "fell through" all the input filters or output filters, respectively.

A message will have been *sifted out* if either (a) an error occurred (error filter), or (b) the message received a reply. The latter is the case if the message eventually lead to the invocation of a method (dispatch filter), or during the reification of the message, a reply was supplied for it (meta filter)

Only one message is allowed to pass through the filtergroup at a time. If a message would like to pass the filtergroup while another is already being filtered by it, the message waits until the other message has left the group;

2. *filtering by a filter.*

If a filter is offered a message to filter, it must decide to accept or to reject the message. This is called *the filter accept test* of the message. If the filter rejects the message, its *reject action* is carried out. Otherwise the filter accepts the message and its *accept action* is carried out.

What the *accept* and *reject actions* actually involve, depends on the filter handler. They will be described in the section 11.6.

If the filter accepts the message, a new target object and a new selector becomes available, which some filters ignore (wait filter), some substitute them in the message (error, dispatch and send filter) and others use it as the receiver and selector of a new message (meta filter);

3. *the filter accept test.*

The message is offered to each filterelement of the its initializer in turn, starting with the left-most, to test if the filterelement accepts it. If it does so, the filter will accept the message, otherwise the message offered to the next-right filterelement. If none of the filterelements in the filterinitializer accepts the message, the filter rejects the message.

4. *the filterelement accept test.*

There are two kinds of filterelements: the inclusionelement $C \Rightarrow M$ and the exclusionelement $C \sim M$ or $C \sim \{M_1, \dots, M_n\}$, in which C is a condition and M, M_1, \dots, M_n are messagepatterns.

a) The inclusion filterelement $C \Rightarrow M$ accepts a message if the condition C evaluates to true and the message matches messagepattern M .

b) The exclusion filterelement $C \sim M$ accepts a message if the condition C evaluates to true and the message does *not* match messagepattern M .

c) The exclusion filterelement $C \sim \{M_1, \dots, M_n\}$ accepts a message if the condition C evaluates to true and the message does *not* match *any* of the messagepatterns M_1, \dots, M_n .

5. *messagepattern matching.*

There are two kinds of messagepatterns: the matchingpattern $[mt.ms]st.ss$ and the signaturepattern $mt.ms$, in which mt is the matching target, ms the matching selector, st the substitution target and ss the substitution selector.

a) A message matches the matchingpattern $[mt.ms]st.ss$ if the receiver rcv of the message matches with the target mt , and if the selector sel of the message matches with the matching selector ms .

- b) A message matches the signaturepattern $mt.ms$ if the message matches with the signature of target mt , and the selector sel of the message matches with the selector ms .
6. *matching the receiver rcv of the message with targetobject mt.*
If mt is the wildcard then the receiver rcv matches, otherwise the rcv matches with mt if and only if the rcv is the same object as mt refers to.
7. *matching the selector sel of the message with selector ms.*
If ms is the wildcard then the message matches, otherwise the message matches if and only if its selector sel is ms .
8. *matching with the signature of mt.*
If mt is the wildcard then the message matches, otherwise the message matches if and only if the signature of the object to which mt refers includes the selector sel .

The table below lists the accept conditions (steps 4a, 5-8) for all possible inclusion filterelements.

TABLE 11-2

Inclusionelement accept condition for a message m

filter-element	accept condition	target returned	select or returned	filter-element	accept condition	target returned	select or returned
$C \Rightarrow [a.b]c.d$	$C \wedge m.rcv = a \wedge m.sel = b$	c	d	$C \Rightarrow [b] d$	$C \wedge m.sel = b$	-	d
$C \Rightarrow [a.b] c.*$	$C \wedge m.rcv = a \wedge m.sel = b$	c	-	$C \Rightarrow [b] *.*$	$C \wedge m.sel = b$	-	-
$C \Rightarrow [a.b] *.d$	$C \wedge m.rcv = a \wedge m.sel = b$	-	d	$C \Rightarrow [b] *$	$C \wedge m.sel = b$	-	-
$C \Rightarrow [a.b] d$	$C \wedge m.rcv = a \wedge m.sel = b$	-	d	$C \Rightarrow [.*] c.d$	C	c	d
$C \Rightarrow [a.b] *.*$	$C \wedge m.rcv = a \wedge m.sel = b$	-	-	$C \Rightarrow [.*] c.*$	C	c	-
$C \Rightarrow [a.b] *$	$C \wedge m.rcv = a \wedge m.sel = b$	-	-	$C \Rightarrow [.*] *.d$	C	-	d
$C \Rightarrow [a.*] c.d$	$C \wedge m.rcv = a$	c	d	$C \Rightarrow [.*] d$	C	-	d
$C \Rightarrow [a.*] c.*$	$C \wedge m.rcv = a$	c	-	$C \Rightarrow [.*] *.*$	C	-	-
$C \Rightarrow [a.*] *.d$	$C \wedge m.rcv = a$	-	d	$C \Rightarrow [.*] *$	C	-	-
$C \Rightarrow [a.*] d$	$C \wedge m.rcv = a$	-	d	$C \Rightarrow [*] c.d$	C	c	d
$C \Rightarrow [a.*] *.*$	$C \wedge m.rcv = a$	-	-	$C \Rightarrow [*] c.*$	C	c	-
$C \Rightarrow [a.*] *$	$C \wedge m.rcv = a$	-	-	$C \Rightarrow [*] *.d$	C	-	d
$C \Rightarrow [*] c.d$	$C \wedge m.sel = b$	c	d	$C \Rightarrow [*] d$	C	-	d
$C \Rightarrow [*] c.*$	$C \wedge m.sel = b$	c	-	$C \Rightarrow [*] *.*$	C	-	-
$C \Rightarrow [*] *.d$	$C \wedge m.sel = b$	-	d	$C \Rightarrow [*] *$	C	-	-
$C \Rightarrow [*] d$	$C \wedge m.sel = b$	-	d	$C \Rightarrow c.d$	$C \wedge m.sel = d \wedge d \in \text{sig}(c)$	c	d
$C \Rightarrow [*] *.*$	$C \wedge m.sel = b$	-	-	$C \Rightarrow c.*$	$C \wedge m.sel \in \text{sig}(c)$	c	-
$C \Rightarrow [*] *$	$C \wedge m.sel = b$	-	-	$C \Rightarrow *.d$	$C \wedge m.sel = d$	-	d
$C \Rightarrow [b] c.d$	$C \wedge m.sel = b$	c	d	$C \Rightarrow d$	$C \wedge m.sel = d$	-	d
$C \Rightarrow [b] c.*$	$C \wedge m.sel = b$	c	-	$C \Rightarrow *.*$	C	-	-
$C \Rightarrow [b] *.d$	$C \wedge m.sel = b$	-	d	$C \Rightarrow *$	C	-	-

No target and selector returned by an exclusionelement

Table 11-3 below lists the accept conditions (steps 4b, 4c, 5-8) for the most of possible exclusion filterelements. In this table, \bullet denotes either $c.d$, $c.*$, $*.d$ or $*.*$. The left column shows filterelements $C \sim M$ with a single message pattern M , while the right column shows a filterelement $C \sim \{M_1, M_2, \dots\}$ with a set of message patterns M_1, M_2, \dots .

Note that an exclusionelement does not return a target or selector if the message matches it.

*Do not specify an exclusionelement containing the wildcard pattern *.*.*

TABLE 11-3

Note also, that if an exclusionelement contains the wildcard pattern **.**, a message will never be accepted by it. It is therefore useless to specify an exclusionelement containing such a wildcard pattern, like *C~>[*.*]••*, *C~>*.**, *C~>{..., [*.*]••, ...}* or *C~>{..., *.* , ...}*.

Exclusionelement accept condition for a message m

filter-element	accept condition	target returned	select or returned	filter-element	accept condition	target returned	select or returned
<i>C~>[a.b]••</i>	$C \wedge (m.rcv \neq a \vee m.sel \neq b)$	-	-	<i>C~>{ [a.b]••,</i>	$C \wedge (m.rcv \neq a \vee m.sel \neq b)$	-	-
<i>C~>[a.*]••</i>	$C \wedge m.rcv \neq a$	-	-	<i>[a.*]••,</i>	$\wedge m.rcv \neq a$		
<i>C~>[* .b]••</i>	$C \wedge m.sel \neq b$	-	-	<i>[* .b]••,</i>	$\wedge m.sel \neq b$		
<i>C~>[* .*]••</i>	false	-	-	<i>[* .*]••,</i>	$\wedge \text{false}$		
<i>C~>c.d</i>	$C \wedge (m.sel \neq d \vee d \notin \text{sig}(c))$	-	-	<i>c.d,</i>	$\wedge (m.sel \neq d \vee d \notin \text{sig}(c))$		
<i>C~>c.*</i>	$C \wedge m.sel \notin \text{sig}(c)$	-	-	<i>c.*,</i>	$\wedge m.sel \notin \text{sig}(c)$		
<i>C~>*.d</i>	$C \wedge m.sel \neq d$	-	-	<i>*.d,</i>	$\wedge m.sel \neq d$		
<i>C~>*.*</i>	false	-	-	<i>*.* , ...}</i>	$\wedge \text{false} \wedge \dots$		

11.5 The signature of an object

The signature of an object is the set of message selectors that the object would accept if all the conditions in the input filters, and in the input filters only, would be true.

DEFINITION 11-1

The signature of an object

- The signature of inner is the set of selectors of the interface methods.*
- The signature of a Sina object a set which includes*
 - 1)the non-wildcard matching selectors ms of the matching patterns [mt.ms]st.ss of the input filters which perform substitution on the message.*
 - 2)the signatures of the objects mt in the signature patterns mt.ms of the input filters which perform substitution on the message.*
- The signature of a Smalltalk object is the set of selectors of the methods defined by its class and superclasses.*

To be specific: the signature of a Sina object is a set that includes (1) the selectors of the interface methods, (2) the non-wildcard matching selectors of matching patterns of input Error filters, (3) the non-wildcard matching selectors of matching patterns of input Dispatch filters, and (4) the signatures of the objects in signature patterns in input Dispatch filters.

As an example of the signature of an object, consider the class Mailbox below, in which anyone can store a letter and only if YouAreAPostman you can retrieve letters from it and access the box itself. The signature of an instance of Mailbox consists of the selectors putLetter, getLetters (the interface methods), emptyBox (matching selectors of the Error filter) and all the selectors defined by the object box (signature of box in Dispatch filter).

EXAMPLE 11-4

```

class Mailbox interface
  internals
    box : Letterbox;
  conditions
    ifYouAreAPostman;
  methods
    putLetter(aLetter : Letter) returns nil;
    getLetters returns Letters;
  inputfilters
    error : Error = { [emptyBox]getLetters, inner.*, box.*};

```

```

delegate : Dispatch = { ifYouAreAPostman => {inner.getLetters, box.*},
                        inner.putLetter};

```

```

end;

```

11.6 Filterhandlers

In this section we describe the various filterhandlers. The filterhandlers implemented in *Sina/st version 3.00* include Dispatch, Error, Send, Wait and Meta. Table 11-4 summarizes the accept action (left) and reject action (right) of each filterhandler. We have indicated if the filter performs the substitution of a new target and selector on the filtered message or if it discards them. For each filtertype it is further marked whether it can be used as an input filter, as an output or both.

TABLE 11-4

Filter handler accept action and reject action.

Dispatch	Substitutions on message	Input
<p>The new target and new selector are substituted to the accepted message, if they are defined; If the receiver of the message became is inner, the method with the same selector as the message will be invoked; Otherwise the message will be offered for filtering to the input filters of the receiver of the message.</p>		The message continues in the next filter.
Error	Substitutions on message	Input Output
<p>The new target and new selector are substituted in the accepted message, if they are defined; Then the message continues in the next filter.</p>		Raises the error 'Reject in ErrorFilter $X::f$ of M ', where X is the class in which this error filter f is defined and M is a representation of the message.
Send	Substitutions on message	Output
<p>The new target and new selector are substituted to the accepted message, if they are defined; If its owning object has no encapsulating object, or if the receiver of the message lies inside its encapsulating object, the message will be offered to the input filters of the receiver of the message. Otherwise the message is offered to the output filters of the encapsulating object.</p>		The message continues in the next filter.
Wait	Substitutions discarded	Input Output
<p>The message continues in the next filter. Note that no substitution are carried out.</p>		<p>The message is blocked, i.e. execution of the calling thread is suspended until the condition on which the message blocked becomes true; Then the message continues in the next filter.</p>
Meta	Used for message to ACT	Input Output
<p>The execution of the calling thread is suspended; The accepted message is reified and becomes an instance of the class <code>SinaMessage</code>; A new thread is created which sends a message with the new selector and the reified message as argument to the new target. This message does not pass through the output filters of the owning object but is offered to the input filters of the new target directly; The sending thread remains suspended until the reified message is dereified—this is if it receives a send, reply, fire or continue message; If the, now active message has a reply—if the reified form was reactivated with a reply message—it will not carry on in the next filter but return the reply to the sender. Otherwise it will be filtered by the following filter.</p>		The message continues in the next filter.

11.6.1 Dispatch filter

A dispatch filter can be used for inheritance and delegation. By declaring a dispatch filter which dispatches to the appropriate internal or external we can accomplish inheritance and delegation, respectively. Though dispatching a message to an internal object can be seen as inheritance, and dispatching to an external object as delegation, there is no distinction between those two: a dispatch filter dispatches a message to another object.

By specifying more target objects, we can accomplish multiple inheritance, cq. delegation. The left to right ordering in the filterinitializer resolves any name conflicts resulting from multiple inheritance.

In [Bergmans94a] and [Bergmans94b] it is further shown how the dispatch filter can be used for *dynamic* inheritance and delegation.

As an example of the dispatch filter, consider the class Manager below. The dispatch filter inheritAndDelegate specifies that this class inherits from class Employee and further delegates messages to the external DepartmentSecretary. If both the classes Employee and Secretary support the method name, then the name method of Employee would be invoked, because it is the first target in the dispatch filter.

```
class Manager interface
  externals DepartmentSecretary : Secretary;
  internals empl : Employee;
  ...
  inputfilters
    inheritAndDelegate : Dispatch = {employee.*, DepartmentSecretary.*};
end;
```

11.6.2 Error filter

An error filter can be applied for checking incoming messages, multiple views and preconditions. In the Stack class (example 4-1A, page 26) we have seen the use of an error filter for checking messages. The use of the error filter for multiple views and preconditions has been described in [Aksit92], [Bergmans94a] and [Bergmans94b].

11.6.3 Send filter

A send filter can only be used as output filter for delivering a message to its receiver. If no output filters are declared for a class, a send filter is inserted automatically by the compiler.

11.6.4 Wait filter

A wait filter can be used for synchronizing messages. A message which is rejected by a wait filter is blocked until the message would be accepted by the wait filter. Synchronization constraints are specified as the conditions in the initializer of a wait filter.

As an example, consider the class BoundedBuffer which allows you to store and retrieve objects using the put and get method, respectively. An object can be stored in the buffer if the buffer is not full, while an object can be retrieved from it if the buffer is not empty. These synchronization constraints are specified by the wait filter bufferSync: a put message will be blocked if the notFull condition evaluates to false. A get message will be blocked if the notEmpty condition evaluates to false.

EXAMPLE 11-5

```
class BoundedBuffer(limit : SmallInteger) interface
  conditions
    notFull;
    notEmpty;
  methods
    put(object : Any) returns nil;
    get returns Any;
```

inputfilters

```

    bufferSync : Wait = {notFull => put, notEmpty => get};
    dispatching : Dispatch = {inner.*};
end; // BoundedBuffer interface

```

The implementation of the BoundedBuffer (shown in example 10-1 A, page 66), establishes that after a `get` message the condition `notFull` becomes true thus releasing a blocked `put` message. A `put` message changes the condition `notEmpty` to true, thus releasing a blocked `get` message.

*Consecutive wait filters
evaluated as if they are a
single wait filter*

There is one additional rule which applies to wait filters and to wait filters only. Subsequent wait filters in the filtergroup (input filters or output filters) are evaluated as if they were a single wait filter having conditions which are combined from every wait filter with an **and**. This has the effect that constraints by earlier wait filters are not forgotten. A message will not be blocked if all the conditions in these consecutive filters evaluate to true. If there is at least one condition which is false, it will be blocked. The message is allowed to resume if all the conditions of the consecutive wait filters are true.

Consider for example the input filters of the class `MemBuffer` which allocates memory to store elements.

class MemBuffer interface

```

...
inputfilters
    memorySync : Wait = {memAvail => put, true => get};
    bufferSync : Wait = {notFull => put, notEmpty => get};
    dispatching : Dispatch = {inner.*};
end;

```

The two wait filters are evaluated as if they had been the single wait filter

```

sync : Wait = {memAvail and notfull => put, true and notEmpty => get};

```

11.6.5**Meta filter**

A meta filter is used for reflection on object interactions. Chapter 12 discusses this filter type and its application more elaborately. The background of the meta filter can be found in [Aksit93].

That concludes the discussion of the filters and the filtering of a message.

12

Message passing semantics

Abstract Communication Type

This chapter describes what can happen when an object sends a message to another object. The first section explains the two steps of message execution: filtering and method invocation. Section 12.2 discusses how a message and its attributes can be accessed during its execution.

In *Sina/st*, a message can execute concurrently with other messages. Concurrent executing messages are described by the notion called *thread*. This is the subject of section 12.3.

During the filtering of a message, a meta filter can *reify* the message. Message reification allows an object to reflect on message communication between objects. Section 12.4 describes the process of message reification. An object which is able to reflect on —to abstract— message communication is known by the term *abstract communication type*, or *ACT* [Aksit89][Aksit93]. An ACT object can use reflection on message communication for constraint checking and controlling object interactions.

The last section in this chapter reveals how the attributes of a reified message can be accessed and changed.

12.1 Sending a message

An object can request a service from another object by sending a message to it. Message passing in the *composition filters object model* follows the request/reply model: after a client object has sent a message —the request— to a server object, the client object waits until it receives a return value for the message —the reply— from the server object.

The server object is entirely responsible for servicing the request: it can decide to handle the request itself (by executing some method), to delegate the request to another object, or even to reject it.

In the Composition Filters Object Model, and thus also in *Sina/st*, the execution of the message involves two steps:

- first, the message is filtered by filters.

If the message leaves the object boundary of the sending object¹, the message will pass through the output filters of the sending object. Then, the message passes through the input filters of the server object. This object can decide (by the use of a dispatch filter)

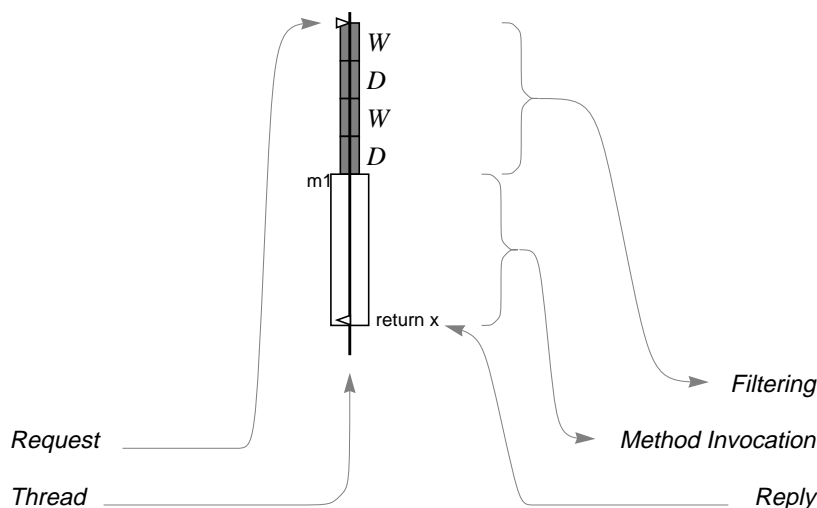
to delegate the message to another object where it is filtered by the input filters of that object.

- eventually, a method is invoked.

The method is invoked if there is a dispatch filter which dispatches the message to inner.

Those two steps are depicted in the diagram of figure 12-1. Similar message execution diagrams are used in the rest of this chapter.

FIGURE 12-1

Execution of a message

In such a message execution diagram, time increases towards the bottom. The start of the message execution, i.e. the issue of a message request, is indicated by a left pointing triangle ▷. The message passes through the filters, shown by grey rectangles ■. In the figure above, we have indicated four filters: two wait filters (W) and two dispatch filters (D). The last filter invokes a method², m1 in this case, which is represented by the larger, white rectangle. The message execution ends when the invoked method returns an object (return x): this reply on the message has been denoted by a right pointing triangle ◁.

Active message

We will call a message which is being executed, an *active message*. We can refer to an active message by the pseudo variable *message*. We explain more on this in the following section.

Thread

After the execution of a message has ended, the following message is executed in a similar way: it is filtered and it invokes a method. The invocation of the method, the white rectangle in the figure above, consists of a sequence of message executions on its own. If we magnify the white rectangle, we could see several message executions one after another, each of them consisting of filtering and method invocation again³. A sequence of such message executions is called a *thread*. The thread will be the topic of section 12.3.

1. This is the case if the message is sent to an external of the sending object, or to the pseudo variables *self*, *server* or *sender*. A message to an internal, an instance variable, a class parameter, a method parameter or local variable, or to the pseudo variables *message* or *inner*, does not pass the output filters, but enters the input filters of the server object directly.
2. This is a dispatch filter. A dispatch filter invokes a method m1 if its filterinitializer contains a filter element like *inner.** or *inner.m1*.
3. Where would this end, you might ask. There are some methods that do not send other messages: they implement primitive operations such as the addition of two numbers.

12.2 The pseudo variable message and class ActiveMessage

In *Sina/st*, the pseudo variable message represents the active message which is an instance of the class `ActiveMessage`. Inside a method, message refers to the message that lead to the invocation of that method. Since condition evaluation is part of the message filtering process, the pseudo variable message inside a condition refers to the message that is currently being filtered.

TABLE 12-1

Accessing the attributes of message.

message.selector	Returns a String which is the selector of the message.
message.numArgs	Returns a <code>SmallInteger</code> , indicating the number of arguments of the message.
message.args	Returns an Array containing the arguments.
message.argsAt(n)	Returns the nth argument.
message.sender	Returns the sender of the message. In a method, this object is also available through the pseudo variable sender.
message.server	Returns the server of the message. In a method, this object is also available through the pseudo variable server.
message.receiver	Returns the receiver of the message.
message.isRecursive	Returns true if the message is sent to self, server, or sender

By sending the appropriate messages to the pseudo variable message, we can access the attributes of an active message. The attributes of a message include:

- its *selector*;
- its *arguments*;
- the *sender*: the object which has sent the message;
- the *server*: the object which has received the message first;
- and its *receiver*: the object which has received the message at this moment. Initially, this attribute is the same as the server, but it becomes another object if a dispatch filter dispatches the message to that object.

Recursive message

An active message is defined as recursive if the message is sent to the pseudo variable self, server or sender. This is used by the default synchronization wait filter (see section 8.1.3, page 48) to allow such messages to pass.

The attributes of an active message can be retrieved, but they cannot be changed. The messages that can be sent to an instance of class `ActiveMessage` have been listed in appendix B.2, page 114. In table 12-1 however, we have indicated the messages often used to access the active message.

12.3 Concurrency with threads

A thread is a sequence of message executions. In *Sina/st*, any number of threads can be created and exist at the same time. In this way, messages (and methods) can execute concurrently. Note that more than one thread can be active—executing concurrently—inside the same object: there are two or more methods which have been invoked and which are executing. If this is not desired, the object must have defined mutual exclusion with the aid of an input wait filter⁴.

4. The *Sina/st* compiler inserts by default for every class an input wait filter that realizes mutual exclusion (see section 8.1.3, page 48). The programmer can override this, and define a more complex synchronization scheme for the class or no synchronization at all.

There are two occasions in which a new thread is created and started:

- upon the invocation of an early return method (see section 9.6, page 58, for the difference between a normal and an early return method),
- when a message is sent to an ACT, upon the reification of a message by a meta filter.

A thread ends when the last message of that thread has been executed: it is the last message of either an early return method or of the ACT method. Note that if such a method contains an infinite loop, the thread will never finish.

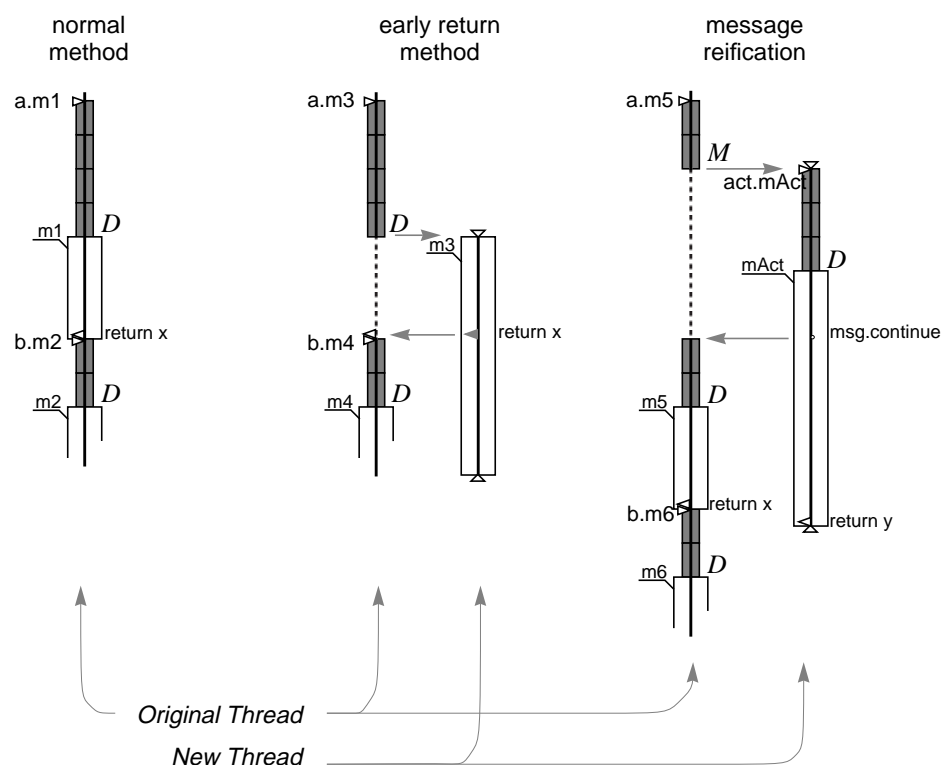
In diagrams, the start of a new thread is represented by the downward pointing triangle (∇). The end of a thread is depicted by an upward pointing triangle (\triangle).

The left diagram of figure 12-2 shows that a normal (non-early return) method is executed in the same thread: the message a.m1 is filtered and the method m1 is invoked. After method m1 has returned an object (return x), the next message (b.m2) is filtered and method m2 is invoked.

The middle diagram shows the execution of an early return method m3. The message a.m3 is filtered as usual. After the last filter, a dispatch filter, a new thread is created for the execution of method m3: The original thread is blocked until method m3 returns an object x. The message now has a reply and the blocked thread (the dashed line -----) resumes by executing the next message, b.m4. After the method has returned a reply, the rest of m3 (the messages following the return) is executed concurrently with the original thread, and thus with message b.m4. The new thread ends when the last message in the early return method m3 has been executed.

FIGURE 12-2

Single thread (left) and new concurrent threads (middle and right).



A new thread is also be created upon the reification of a message by a meta filter (the right most diagram in figure 12-2). Message reification and dereification is explained in the next section. For now, we can see that the message act.mAct executes in the new thread concurrently with method m5 in the original thread.

12.4 Message reification and dereification

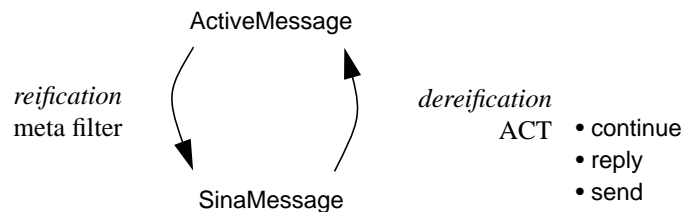
Message *reification* is the process of converting an active message into a first-class object, a so called *reified message*. The opposite process is called *dereification* and converts a reified message back into an active message. Reification suspends the execution of a message, while dereification reactivates it. A reified message is an instance of the class `SinaMessage` and, as we have seen in section 12.2, an active message is an instance of class `ActiveMessage`.

The reification process takes place in a meta filter: if a meta filter accepts a message, it will reify the message. Then, the meta filter offers the reified message to an *abstract communication type* (ACT) object [Aksit93]. The ACT object is an arbitrary internal or external object, that is, it is an instance of some *Sina/st* class.

A reified message will be reactivated if it receives one of the dereification messages `continue`, `reply` or `send`. The ACT object is responsible for sending one of these messages to the reified message (figure 12-3).

FIGURE 12-3

Reification and dereification of a message.



12.4.1 Message reification

Reification of messages is realized in *Sina/st* by a meta filter. This is the only location where active messages can be reified. If an active message is accepted by a meta filter, it will be reified—become an instance of class `SinaMessage`—and used as the single argument of a new message. The new message and its receiver (an ACT object) are specified as the substitution part of a filterelement in the filterinitializer of that meta filter. A new thread is created and then this message is sent to the ACT. The thread of the original, but now reified message is blocked until the ACT dereifies the message.

We illustrate this with an example. Consider, for instance, the meta filter `divcheck` of the class `NumericProcessor` below. This class provides the standard arithmetic operations, such as addition, subtraction, multiplication and division. Division by zero for the `divide` and `modulo` operations is checked by the internal object `divisionCheck`. This object is an instance of class `DivisionACT` which we encounter later when we describe dereification.

EXAMPLE 12-1 A

```

class NumericProcessor interface
  internals divisionCheck : DivisionACT;
  methods
    add(n1, n2 : Number) returns Number;
    subtract(n1, n2 : Number) returns Number;
    multiply(n1, n2 : Number) returns Number;
    divide(n1, n2 : Number) returns Number;
    modulo(n1, n2 : Number) returns Number;
  inputfilters
    divcheck : Meta = { [divide]divisionCheck.divBy0,
                        [modulo]divisionCheck.divBy0 };
    calculate : Dispatch = { inner.* };
  end; // NumericProcessor interface

class NumericProcessor implementation
  methods
    add(n1, n2 : Number)          begin return n1 + n2    end;
    subtract(n1, n2 : Number)      begin return n1 - n2    end;
  
```

```

multiply(n1, n2 : Number)    begin return n1 * n2    end;
divide(n1, n2 : Number)      begin return n1 / n2    end;
modulo(n1, n2 : Number)      begin return n1 mod n2   end;
end; // NumericProcessor implementation

```

The filter initializer of the meta filter `divcheck` has two filterelements which specify that only message with the selector `divide` and `modulo` is accepted and, consequently, reified. The substitutionpart of both elements define that, as a result, the message `divBy0` is sent to the internal `divisionCheck`.

EXAMPLE 12-1 B

```

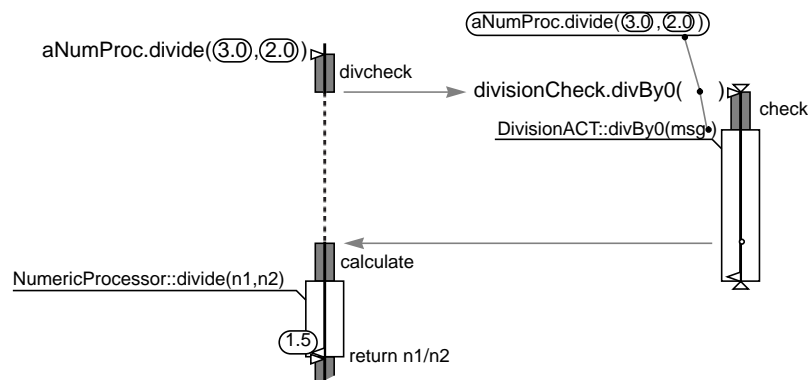
main
  temps
    aNumProc : NumericProcessor;
    result : Number;
  begin
    result := aNumProc.divide(3.0,2.0);
    result := aNumProc.divide(4.0,0.0);
    result := aNumProc.add(5.0, 6.0);
  end

```

Figure 12-4 shows what happens upon the reification of the first `divide` message of the `main` method of example 12-1 B. The message which has 3.0 and 2.0 as arguments, is sent to the object `aNumProc`, an instance of class `NumericProcessor`. This active message is shown as `aNumProc.divide(3.0,2.0)` in the diagram below.

FIGURE 12-4

Reification of a message



First, the message is filtered by `aNumProc`'s input filter `divcheck`. This meta filter accepts it and converts it to the reified message object `aNumProc.divide(3.0,2.0)`, an instance of class `SinaMessage`. This message object is the single argument of the message `divisionCheck.divBy0()`, which is executed in a newly created thread. The original thread is blocked until the reified message is dereified. The `divisionCheck.divBy0()` message is sent to the `divisionCheck` object and is filtered by its input filters as normal. As a result, `DivisionACT`'s method `divBy0` is invoked which dereifies its argument `msg`, the message object `aNumProc.divide(3.0,2.0)`. It turns into the active message `aNumProc.divide(3.0,2.0)` again which then continues in `NumericProcessor`'s next filter, the dispatch filter `calculate`. This filter likewise accepts the message and after that, `NumericProcessor`'s method `divide` is invoked. This method eventually returns the `Float` object 1.5 as result of the message.

12.4.2

Message dereification

Reactivating a reified message—an instance of class `SinaMessage`—is accomplished by sending an appropriate *dereification message* to it. There are three of such dereification messages which are all defined by the class `SinaMessage` (see also section 12.5):

- `continue`, which causes the reified message to be reactivated: it continues its normal course, that is, it is filtered by the filters following the reifying meta filter and eventually invoke a method which returns a value;

- `reply(aReplyObject)`, which causes the message to become active again, and then provides `aReplyObject` as the return value of the message, *just as if* a method had been invoked that returned `aReplyObject`. In this case, the active message is not filtered by the filters following the reifying meta filter and no method is invoked;
- `send`, which causes the message to be reactivated, like the `continue` message, except when the message has received a return value⁵, the active message becomes reified again. Unlike `continue`, the dereification message `send` makes it possible to access the return value of the active message.

After a `send`, the reified message must be reactivated again, either with a `reply` which can supply a different return value for the message, or with `continue` which does not change the return value of the message.

EXAMPLE 12-1 C

```

class DivisionACT interface
  comment
    'This class is an ACT that checks division operations of an instance
    of class NumericProcessor';
  methods
    divBy0(msg : SinaMessage) returns nil;
  inputfilters
    check : Dispatch = { inner.* };
end; // DivisionACT interface

class DivisionACT implementation
  methods
    divBy0(msg : SinaMessage)
      temps NAN : Number basicNew;
      begin
        if msg.numArgs = 2
          then
            if msg.argsAt(2) = 0
              then msg.reply(NAN); return
            end
          end;
          msg.continue; return
        end;
      end; // DivisionACT implementation

```

We continue with the `NumericProcessor` example to show the use of these dereification messages. Division by zero for its `divide` and `modulo` operations was checked by an instance of class `DivisionACT` (example 12-1 c). This class provides the `divBy0` method which is invoked after a `divide` or `modulo` message has been reified by `NumericProcessor`'s `divcheck` meta filter. This reified message is then passed as the argument `msg` to the `divBy0` method. Note that in the `divBy0` method the pseudo variable `message` refers to the message `divisionCheck.divBy0()` sent by the meta filter, whereas the argument `msg` refers to the reified message.

Dereification with continue

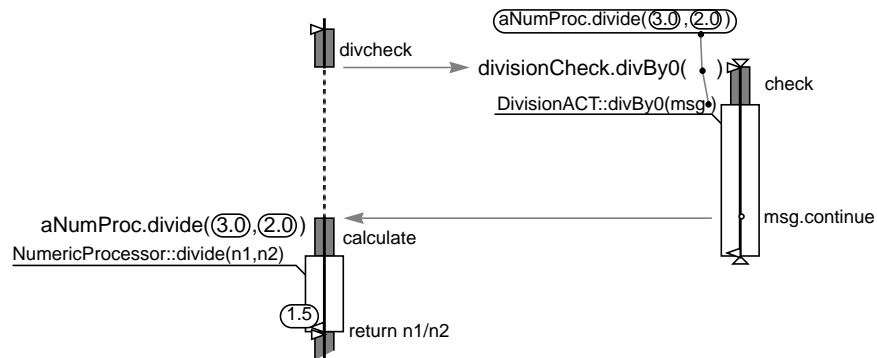
Now, let us follow the message `aNumProc.divide(3.0,2.0)` from example 12-1 B inside the class `DivisionACT`. It has been reified to `(aNumProc.divide(3.0,2.0))` and now is the argument `msg` of the `divBy0` method. This method checks if `msg` has two arguments and, if so, whether its second argument is zero. In this case, the second argument is 2.0 and therefore the `divBy0` method dereifies the message by sending the `continue` message to it. As we can see in figure 12-5, it continues in the dispatch filter and invokes the `divide` method which calculates the result. After the `divBy0` method has dereified the message, it executes the `return` statement. This ends the method and also the thread in the `DivisionACT`. However, the method could have done more operations between the message dereification and the

5. The return value is normally supplied if a method has been invoked which then returns some object, but it could also have been provided by another reification followed by a dereifying reply message.

end of the method. Those actions would have been carried out concurrently with the dereified message.

FIGURE 12-5

Dereification of a message with continue



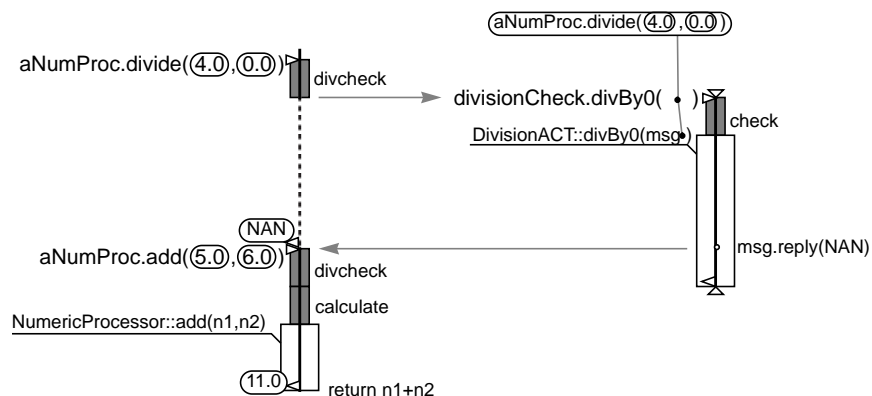
Dereification with reply

The `divBy0` method also uses the message reply to dereify a message. If the second argument of the reified message is zero, `DivisionACT` has detected division by zero and will supply the `Number` object `NAN` —Not A Number— as reply of the message, instead of having the result calculated by `NumericProcessor`'s `divide` or `modulo` operation. The following figure shows the route of the second divide message of example 12-1 which has 4.0 and 0.0 as arguments. Like the first message, the `aNumProc.divide(4.0, 0.0)` message is reified by the `divisioncheck` meta filter and becomes the argument `msg` of the `divBy0` method. Since the second argument of `msg` now is zero, the method supplies `NAN` as reply for the reified message `aNumProc.divide(4.0, 0.0)`. The execution of the now dereified message has finished because it has received a return value: it has not been filtered by the `NumericProcessor`'s dispatch filter `calculate` and no method has been invoked.

Subsequently, the `add` message (the third message of example 12-1 B) is executed. This message is not reified (it is rejected by the meta filter, since its selector is neither `divide` nor `modulo`), but invokes `NumericProcessor`'s `add` method at once.

FIGURE 12-6

Dereification of a message with reply



The third way of reactivating a reified message was by sending it a `send` message. We illustrate this dereification message with the class `LoggingProcessor` which offers the same functionality as the `NumericProcessor` class from example 12-1, but additionally logs every message it receives. Example 12-2 A shows the interface and implementation of this class. The logging of messages is taken care of by the internal logger which is an instance of class `LogACT`. This class is able to store messages and also provides the functionality to display them. The meta filter `log` reifies every message and then present it to logger. The internal `np` provides the behavior of class `NumericProcessor`. With the dispatch filter `disp` its behavior is inherited as well as the behavior of class `LogACT`. Note that the implementation of class `LoggingProcessor` is empty because this class does not define

functionality itself but only composes behavior from the classes NumericProcessor and LogACT.

EXAMPLE 12-2 A

```
class LoggingProcessor interface
  internals
    np : NumericProcessor;
    logger : LogACT;
  inputfilters
    log : Meta = {[*. *]logger.log};
    disp : Dispatch = {np. *, logger.*};
end; // LoggingProcessor interface

class LoggingProcessor implementation
end; // LoggingProcessor implementation
```

Example 12-2 B presents the class LogACT. This class logs messages by storing each reified message together with its return value in the instance variable `loggedMessages`. Every entry in this `OrderedCollection` is a two-element Array: the first element contains the message and the second element holds the return value of the message. The method `showLog` displays the logged messages. The `log` method stores its argument `msg` (a reified message) in a new entry. It then dereifies this message with `send` and waits until `send` returns the return value of the message. This return object is stored in the entry as well. At this point, the message again is reified and therefore has to be restarted with `continue` in order to return the return object to the sender of the message. Finally, the `log` method adds the entry to the `loggedMessages`. Both methods have been made available at the interface by the dispatch filter `disp`.

EXAMPLE 12-2 B

```
class LogACT interface
  methods
    log(msg : SinaMessage) returns nil;
    showLog returns nil;
  inputfilters
    disp : Dispatch = {inner.*};
end; // LogACT interface

class LogACT implementation
  instvars loggedMessages : OrderedCollection;
  methods
    log(msg : SinaMessage)
      temps entry : Array(2);
      begin
        entry.at:put:(1,msg);
        entry.at:put:(2,msg.send);
        msg.continue;
        loggedMessages.add:(entry)
      end;
    showLog
      temps i : SmallInteger; entry: Array(2);
      begin
        for i := 1 to loggedMessages.size
          begin
            entry := loggedMessages.at:(i);
            self.print(entry.at:(1));
            self.print(' returned ');
            self.println(entry.at:(2))
          end;
        end
      end
end; // LogACT implementation
```

EXAMPLE 12-2 C

```
main
  temps
```

```

aLogProc : LoggingProcessor;
result : Number;
begin
  result := aLogProc.divide(3.0,2.0);
  result := aLogProc.divide(4.0,0.0);
end

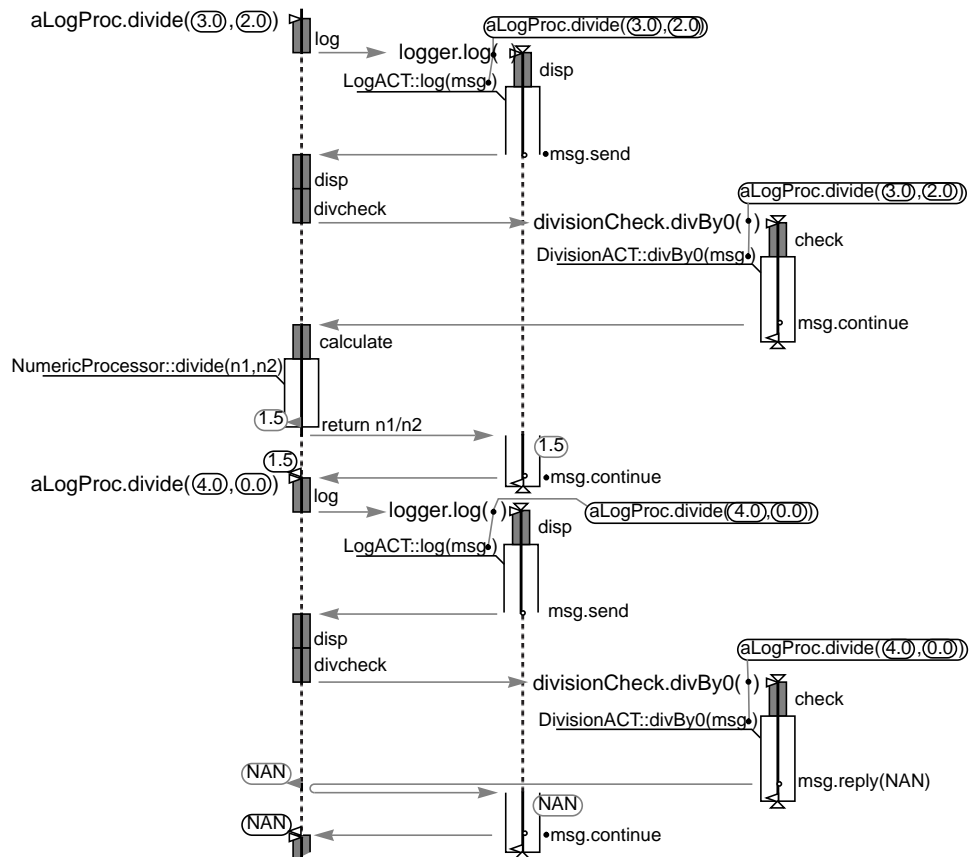
```

Dereification with send

We now illustrate the dereification message send with example 12-2 c. The trace of the messages in this main method is shown in figure 12-7. The first message `aLogProc.divide(3.0,2.0)` is reified by `LoggingProcessor`'s meta filter `log`. The logger object gets control over the presently reified message `aLogProc.divide(3.0,2.0)` and dereifies it with `send`. The thread in the logger blocks until the message receives a return value. The message continues in `LoggingProcessor`'s `disp` filter which dispatches it to the `NumericProcessor` instance `np`. There, the meta filter `divcheck` reifies the message a second time and offers it to the `DivisionACT` instance. Since the second argument of the message is not zero, the `divisionCheck` object decides to dereify it with `continue`. The message then continues in `np`'s dispatch filter `calculate`. This invokes the `divide` method which returns the `Float` instance 1.5 as result of the message. Because the message `aLogProc.divide(3.0,2.0)` now has a return value, control is transferred back to the logger object which resumes its execution by returning the reply of that message, the object 1.5, as the result of the message `send`. The logger object stores this reply object and then dereifies the message `aLogProc.divide(3.0,2.0)` with `continue`. At this point, the message returns with the `Float` object 1.5 in main and has finished its execution.

FIGURE 12-7

Dereification of a message with send



The second message of example 12-2 c, the message `aLogProc.divide(4.0,0.0)`, follows up to the `divisionCheck` object the same course as the previous. This object however, discovers a division by zero condition and supplies the `NAN` object as result for the message. Since the message now has a return value, further filtering is canceled and control is transferred back to the logger object. This object resumes its execution, stores

the reply object NAN and then dereifies the message `(aLogProc.divide(4.0,0.0))` with `continue`. The messages subsequently finishes in main by returning NAN.

12.4.3 Multiple reifications

During the filtering of a message, the message can be reified several times if it is accepted by different meta filters. In the previous section we have already encountered this: the two messages of example 12-2 were reified by `LoggingProcessor`'s log filter and then by `NumericProcessor`'s divcheck filter.

ACTs that dereified a message with `continue` or `reply` do not regain control over the message when it receives a reply. With the dereification message `send` however, the ACT gains control over the reified message and its reply value as soon as the message receives a reply. This reply is either a result of a method invocation or supplied by another ACT with the dereification message `reply`.

It is possible that several ACTs are waiting for the reply on a message because they all dereified the message with `send`. When the message receives a reply, the ACTs will gain control over the message in the reverse order in which they were invoked by respective meta filters. As soon as the message receives a reply, the ACT which was invoked last is the first to gain control over the message and its reply value. This ACT must dereify the message with `continue` or `reply`. Subsequently, the last but one ACT gains control over the message and its reply. This ACT then dereifies the message, and so on, until first ACT dereifies the message.

The following example shows the effect of this. The class `DemonProcessor` extends the behavior of class `LoggingProcessor` by altering the return value of a message. The class `DemonACT` is responsible for this modification. The `change` method adds one to the result of a message only if this result is a `Number`. Otherwise, the result of the message is not changed (example 12-3 A).

EXAMPLE 12-3 A

```

class DemonACT interface
  externals
    Number : Class;
  methods
    change(msg : SinaMessage) returns nil;
  inputfilters
    changelt : Dispatch = {inner.*};
end; // DemonACT interface

class DemonACT implementation
  methods
    change(msg : SinaMessage)
      temps result : Any;
      begin
        result := msg.send;
        if result.class.inheritsFrom:(Number)
          then msg.reply(result+1)
          else msg.continue
        end
      end;
end; // DemonACT implementation

```

The class `DemonProcessor` composes the behavior of the class `LoggingProcessor` and the class `DemonACT`. A message sent to a `DemonProcessor` instance is logged with its correct result by the `LoggingProcessor` instance `lp`. If the result of the message is NAN, it is left unchanged, but if the result is a true number, the `DemonACT` instance `demon` adds one to it (example 12-3 B).

EXAMPLE 12-3 B

```

class DemonProcessor interface

```

EXAMPLE 12-3 C

```

internals
    demon : DemonACT;
    lp : LoggingProcessor;
inputfilters
    demon : Meta = {[*. *]demon.change};
    doit : Dispatch = {lp.*};
end; // DemonACT interface

class DemonProcessor implementation
end; // DemonProcessor implementation

main
    temps
        aDemProc : DemonProcessor;
        result : Number;
    begin
        result := aDemProc.add(3.0,2.0)
    end; // main

```

FIGURE 12-8 Multiple reifications

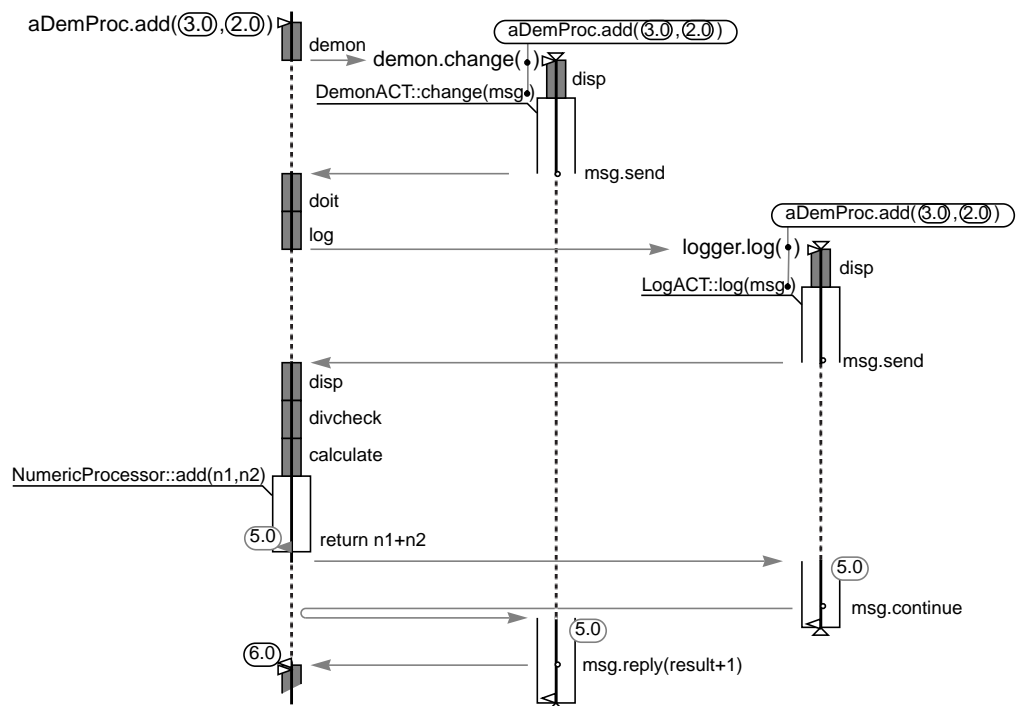


Figure 12-8 shows the flow of the message of example 12-3 c. The message `aDemProc.add(3.0, 2.0)` is accepted by `DemonProcessor`'s first filter, the meta filter `demon`. The message is reified and offered to the `DemonACT` instance `demon`. This object sends the message which continues through `aDemProc`'s dispatch filter `doit`. This filter dispatches it to the `LoggingProcessor` instance `lp`. The meta filter `log` of this object reifies the message a second time and offers it to the `LogACT` instance `logger`. The `logger` object dereifies the message with `send` causing it to continue in dispatch filter `disp`, the meta filter `divcheck` (which does not reify the message) and the dispatch filter `calculate`. Then, `NumericProcessor`'s `add` method is invoked which returns the `Float` object `5.0` as result of the message. Since the message now has a reply, the control is transferred back to the `logger` object which was the last to send the message. The `logger` object logs the message and its return value `5.0` and reactivates the message with `continue`. The control will now be

transferred back to the demon object because it was the previous object which sent the message (with send). This object changes the reply of the message to 6.0. As there are no other ACTs which have sent (send) the message, this result for the message is eventually returned in main.

If an ACT sends one of the dereification message continue, reply or send to a message which it already had dereified with continue or reply, this is semantically incorrect⁶. This is because the ACT gives up the control on the message after it has sent continue or reply to it. With the dereification message send, the ACT gains control on the message once it receives a reply. After a send, the message must be dereified once more with continue or reply. A send followed by a send however, is also incorrect. In the current version of *Sina/st*, version 3.00, sending one of the dereification messages continue, reply or send after a continue or reply has no effect. Neither has sending a send after a send.

12.4.4 Writing ACT methods

In this section we give some hints and tips on writing methods which handle reified messages. We call a method which handles a reified message an ACT method. An ACT object can be an instance of any class which can, apart from one or more ACT methods, have other methods as well.

- an ACT method must have one and only one argument, declared as an instance of class `SinaMessage`. This argument refers to the reified message;
- the pseudo variable `message` in an ACT method refers to the message which was sent by the meta filter, while the argument of the method refers to the reified message;
- it is important that message is dereified before ACT method finishes, because the message would otherwise remain reified forever. Thus, a message must be dereified with continue, reply or send, and after a send the message must be dereified a second time with continue or reply;
- it is not necessary to do an early return in an ACT method, but
- there is a possibility of deadlock if a message is dereified with send. If the thus dereified message does not return a value (for instance, through an infinite loop), the send in the ACT will wait on the return from the message forever. Further assume, that the ACT has protected its data with mutual exclusion by a wait filter (for instance, the `defSync` filter which is inserted by the compiler automatically). Since there is still a method active (viz. the ACT method), new messages to the ACT is blocked forever by the this wait filter.

There is a solution for this. You can use an early return in the ACT method in order to 'free' the ACT, but beware of protecting the data of the ACT that could be accessed after the early return. This could be solved by accessing the data of the ACT through self-calls.

We could redesign the class from `LogACT` from example 12-2 B to prevent the deadlock situation from happening. The result has been presented in example 12-4. The `log` method now performs an early return and, after it has received the result of the reified message, it sends to itself the message `logMessageAndReply`. The corresponding method creates an entry for the message and its reply and stores this.

EXAMPLE 12-4

```
class LogACT interface
  methods
    log(msg : SinaMessage) returns nil;
    logMessageAndReply(msg : SinaMessage; result : Any) returns nil;
    showLog returns nil;
  inputfilters
    disp : Dispatch = {inner.*};
```

6. To be more precise: it is semantically incorrect if in the same ACT thread, a message which was dereified with continue or reply, is dereified a second time.

```

end; // LogACT interface

class LogACT implementation
  instvars loggedMessages : OrderedCollection;
  methods
    log(msg : SinaMessage)
      temps result : Any;
      begin
        return; // early return
        result := msg.send;
        msg.continue;
        self.logMessageAndReply(msg, result)
      end;
    logMessageAndReply(msg : SinaMessage; result : Any)
      begin
        temps entry : Array with:with:(msg,result);
        begin
          loggedMessages.add:(entry)
        end;
        showLog // ... as in example 12-2 B
      end; // LogACT implementation

```

12.5 The class SinaMessage

The class `SinaMessage` describes a reified message, i.e. a first-class representation of a message. As we have seen in section 12.4.1, a meta filter creates an instance of this class if it reifies an active message. This reified message is called a *suspended* message, because it represents a message of which the execution has been suspended. Dereifying a suspended message *reactivates* the message.

We can also create an instance of class `SinaMessage` by declaring a variable of this class, or by copying another reified message. The reified message created in this way has not been active before and we call such a message an *idle* message. Dereifying an idle message activates the message for the first time. This means that the message actually is sent: it is filtered and eventually invokes a method. During filtering, some meta filter could reify the message again and suspend its execution.

The attributes of a reified message are the same as those of an active message (section 12.2). They include its *selector*, *arguments*, *sender*, *server* and *receiver*. The attributes of an active message can only be retrieved, but if the message is reified, they can be modified too. The class `SinaMessage` (appendix B.3, page 115) provides, in addition to the attributes accessing methods of class `ActiveMessage` (appendix B.2, page 114), also methods for changing, dereification and copying. They have been listed in the following table.

TABLE 12-2 Accessing the reified message `aMsg`.

<code>aMsg.selector</code>	Returns the selector of <code>aMsg</code> .
<code>aMsg.numArgs</code>	Returns the number of arguments of <code>aMsg</code> .
<code>aMsg.args</code>	Returns an Array containing the arguments.
<code>aMsg.argsAt(n)</code>	Returns the <i>n</i> th argument.
<code>aMsg.sender</code>	Returns the sender of <code>aMsg</code> .
<code>aMsg.server</code>	Returns the server of <code>aMsg</code> .
<code>aMsg.receiver</code>	Returns the receiver of <code>aMsg</code> .
<code>aMsg.setSelector(newSelector)</code>	Replace the selector by the String <code>newSelector</code> . Returns <code>nil</code> .
<code>aMsg.setArgs(argArray)</code>	Replace the argument by the Array object <code>argArray</code> . Returns <code>nil</code> .

TABLE 12-2

Accessing the reified message aMsg.

aMsg.argsAtPut(n, newArg)	Replace the nth argument by the object newArg. Returns nil.
aMsg.setSender(newObj)	Replace the sender by the object newObj. Returns nil.
aMsg.setServer(newObj)	Replace the server by the object newObj. Returns nil.
aMsg.setReceiver(newObj)	Replace the receiver by the object newObj. Returns nil.
aMsg.copy	Returns a copy of aMsg which has the same selector, server, receiver and arguments, but its sender is undefined (nil).
aMsg.continue	Dereify aMsg: it is activated for the first time if aMsg is an idle message, or reactivated if it is a suspended message. Returns nil.
aMsg.fire	Same as aMsg.continue: it (re)activates aMsg.
aMsg.reply(anObject)	Dereify aMsg and supply anObj as return value for the message. Returns nil.
aMsg.send	Dereify aMsg and wait until the message receives a reply. Returns this reply object.
aMsg.sendContinue	Same as aMsg.send; aMsg.continue

If we send the dereification message `send` to a reified message, `send` will wait until the message receives a reply. The two other dereification messages `continue` and `reply` do not wait for a reply on the message. In *Sina/st version 3.00* however, dereification of an idle message with `continue` does wait for the reply.

Copying a reified message creates a new instance of class `SinaMessage` which has the same selector, arguments, server and receiver as its origin, but its sender is undefined (nil). This instance is an idle message which can be made active by sending one of the dereification messages to it.

A variable which has been declared as an instance of class `SinaMessage` has all its attributes undefined (nil). Before dereifying —actually sending— this idle message, at least the selector, the arguments and its server must be assigned. The following example shows the usage of a `SinaMessage` variable to create and send a message equivalent to `3.0.divide(2.0)`.

EXAMPLE 12-5

```

main
  temps
    msg : SinaMessage;
    args : Array(1);
    result : Number;
  begin
    // The following statements give the same answer as
    //   result := 3.0.divide(2.0);
    msg.setSelector('divide');
    msg.setServer(3.0);
    args.at:put(1,2.0);
    msg.setArgs(args);
    result := msg.send;
  end

```



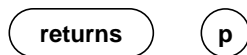

PART iii

Appendices

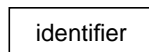
Sina/st Syntax Diagrams

A.1 Conventions

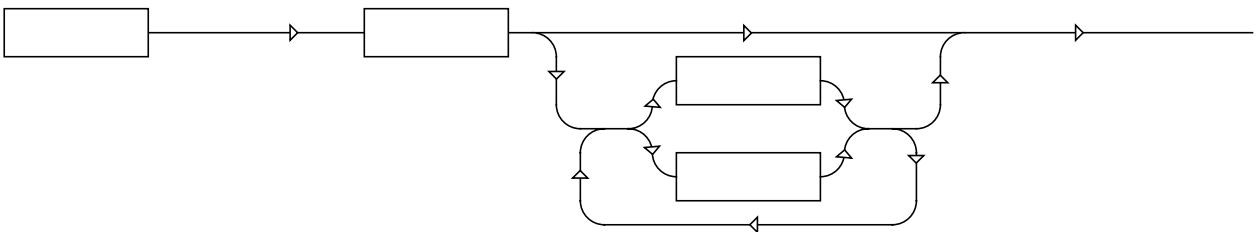
A terminal is represented by a rounded rectangle or a circle with the symbol representing the terminal inside. For instance:



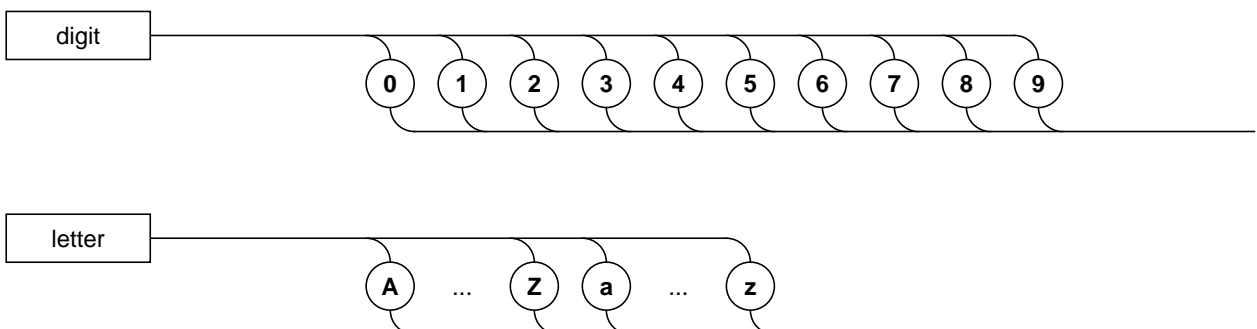
A nonterminal is represented by a rectangle with the name of the nonterminal inside. For instance:

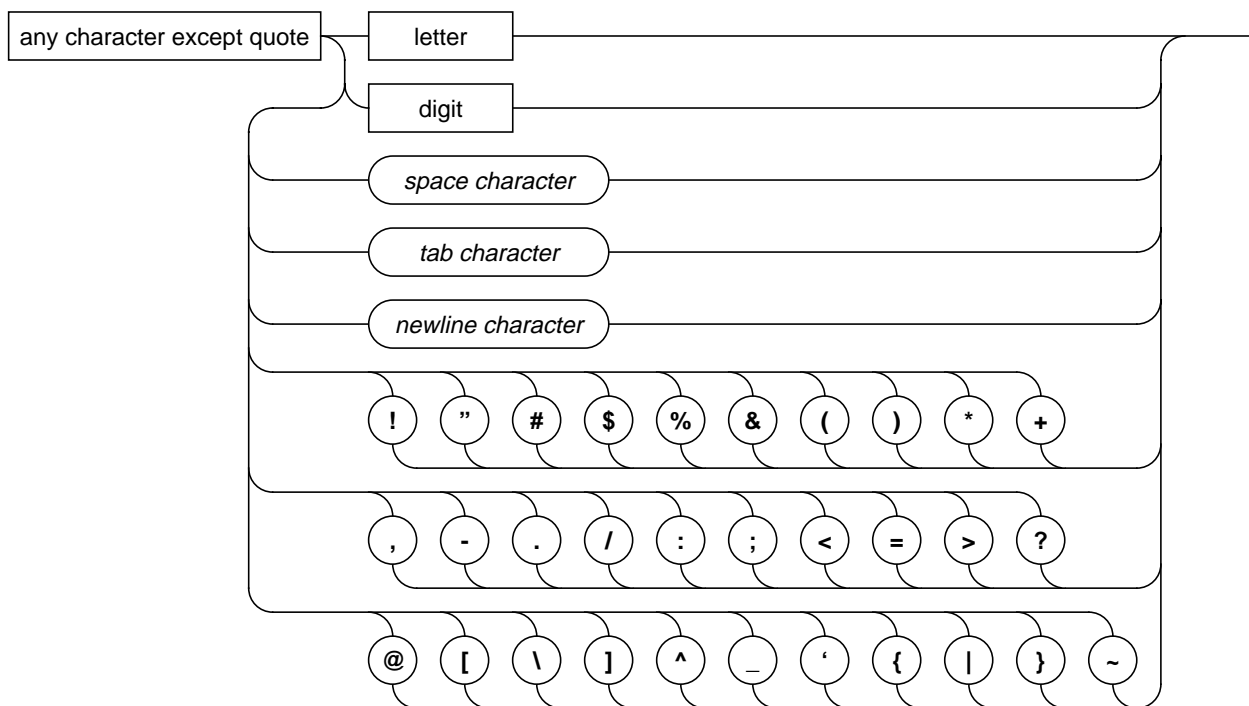


The syntax of a nonterminal is shown with the nonterminal on the far left of the page and a path originating from it and ending on the far right of the page. The path contains terminals and nonterminals and can be seen as a railroad: to find a derivation for the nonterminal you follow the path originating from it, choose a direction at a splitting, pass through stations – terminals and nonterminals, possibly even more than once– until you reach the end of the line. The line segments between two stations or junctions can be travelled only in one way. The example below shows the directions are indicated with an arrow, though they are omitted in the diagrams in the rest of this appendix.

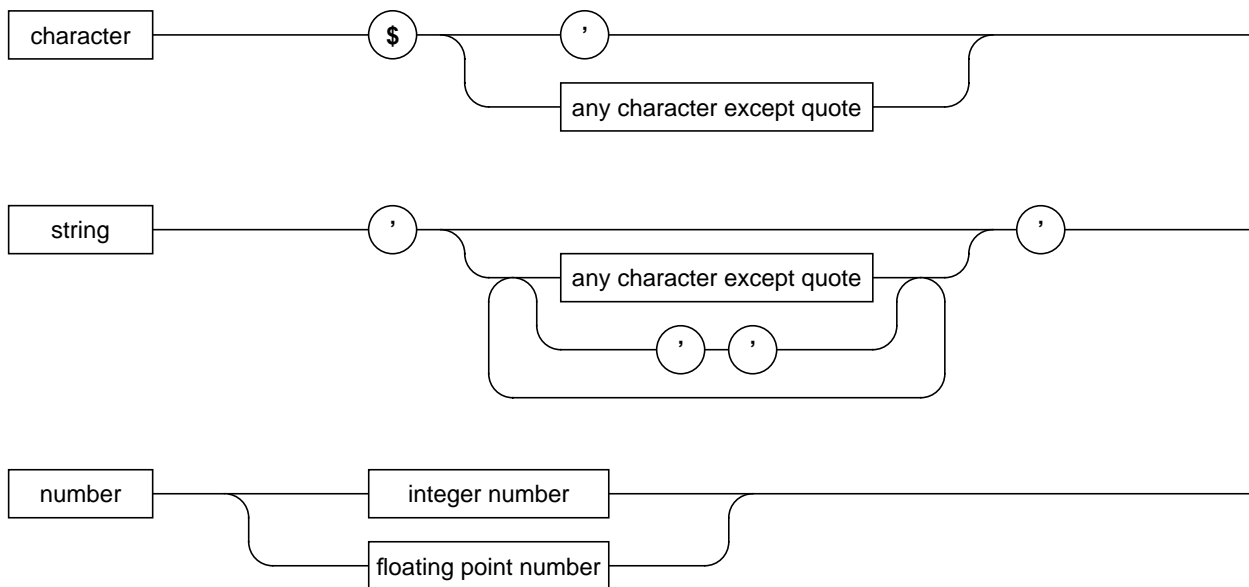


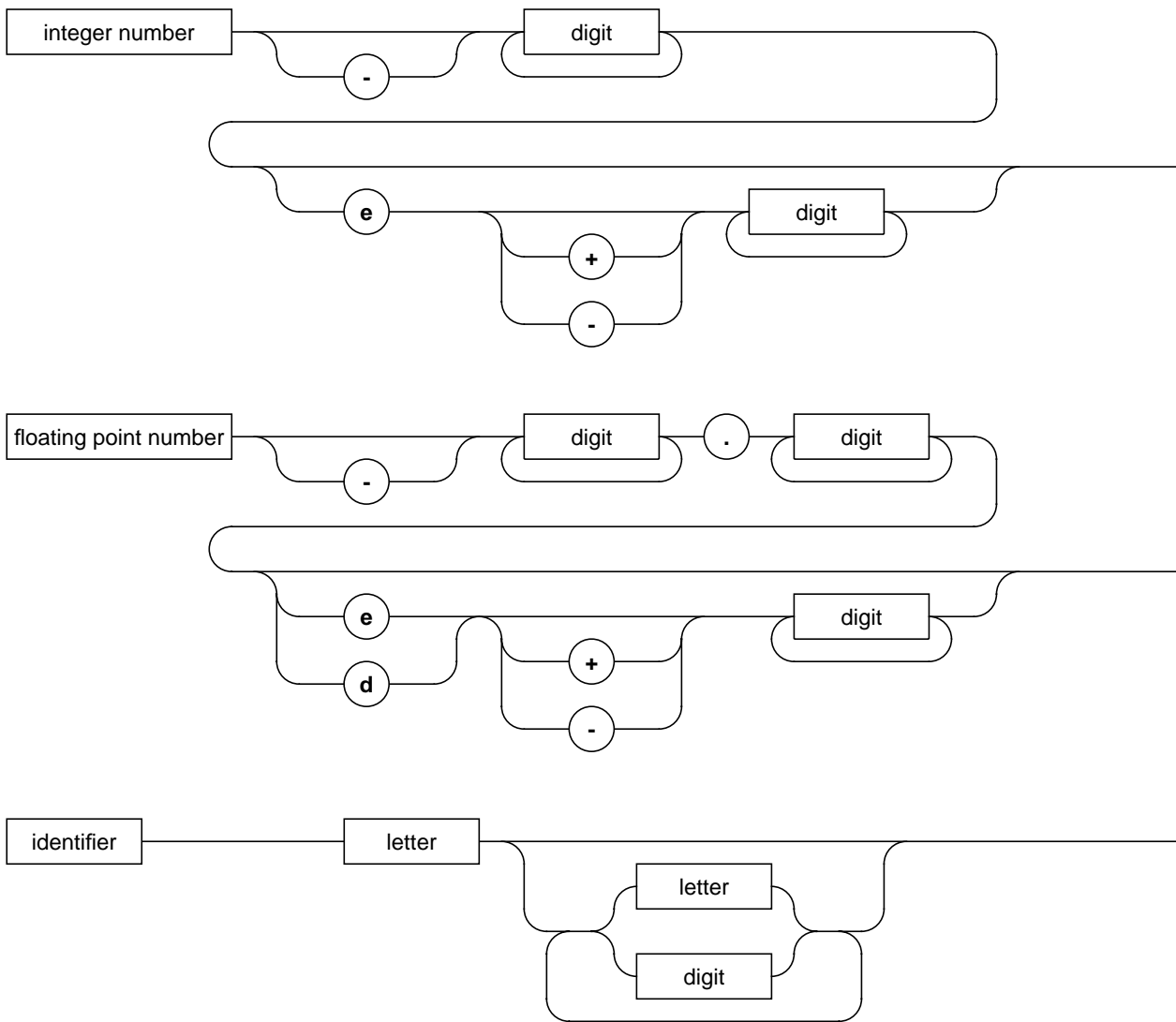
A.2 Denotations



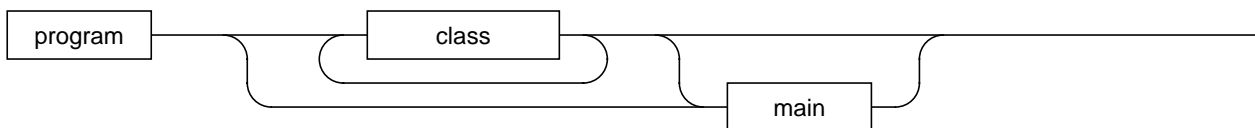


A.3 Literals

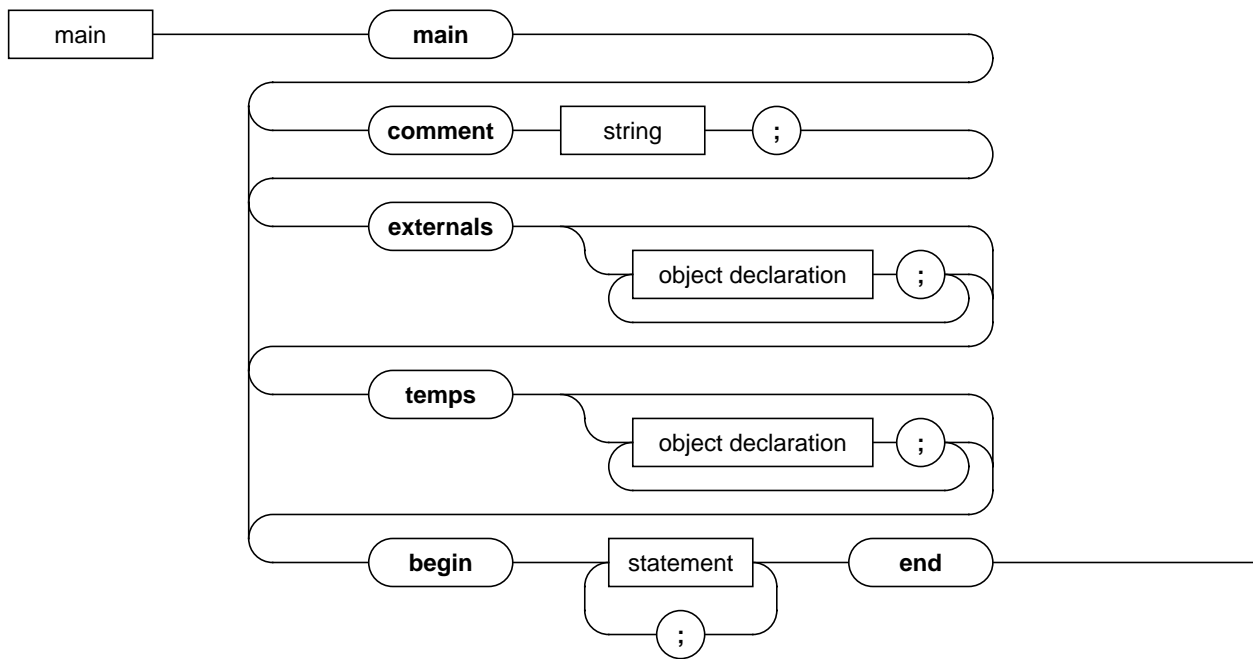




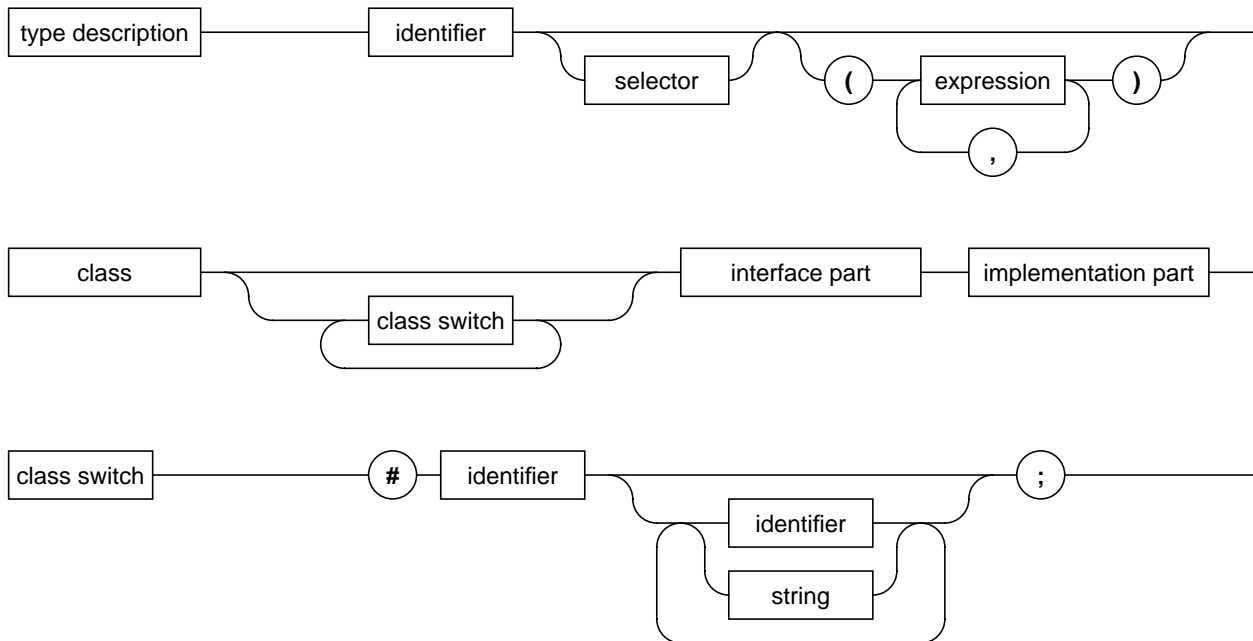
A.4 Program

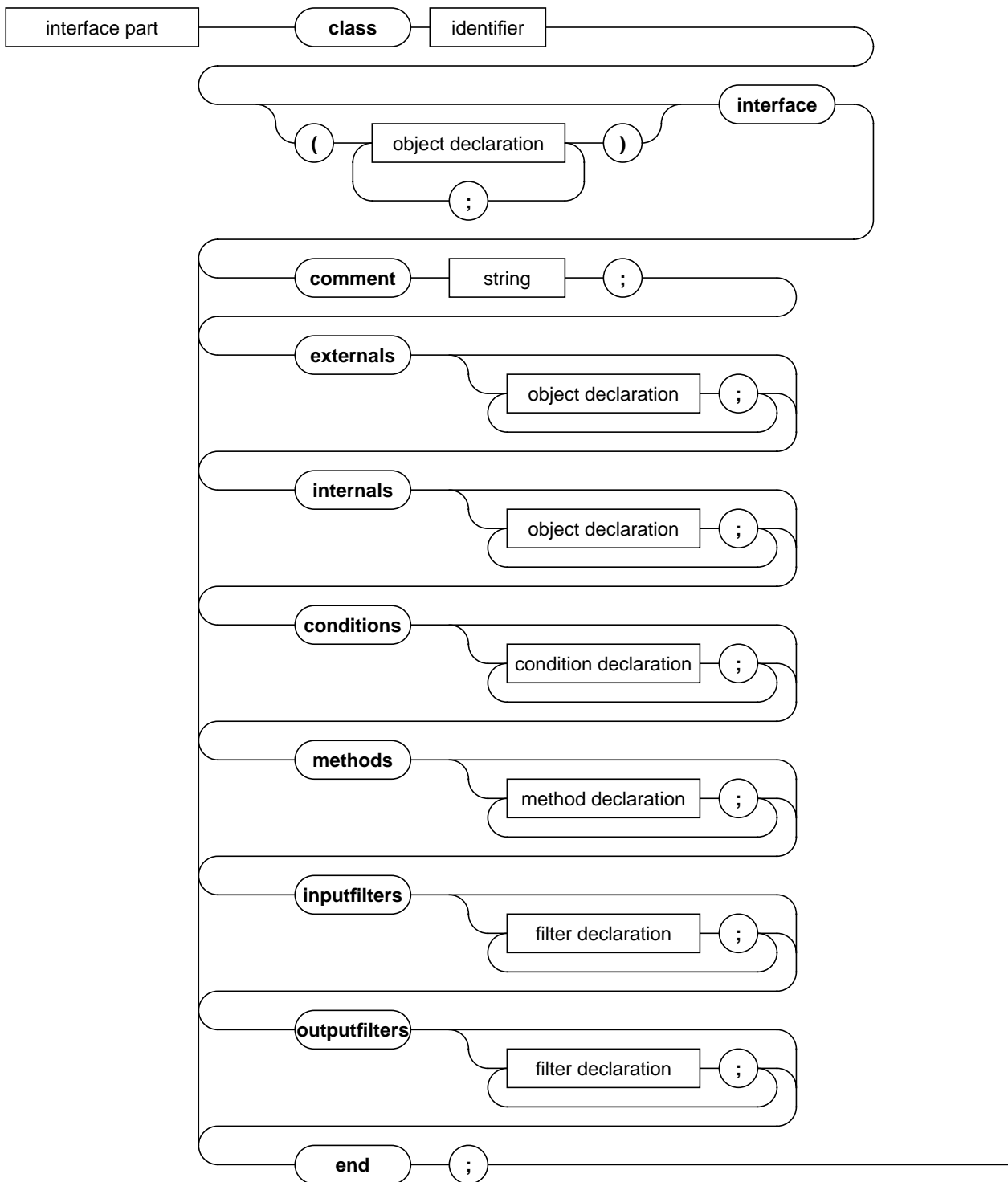


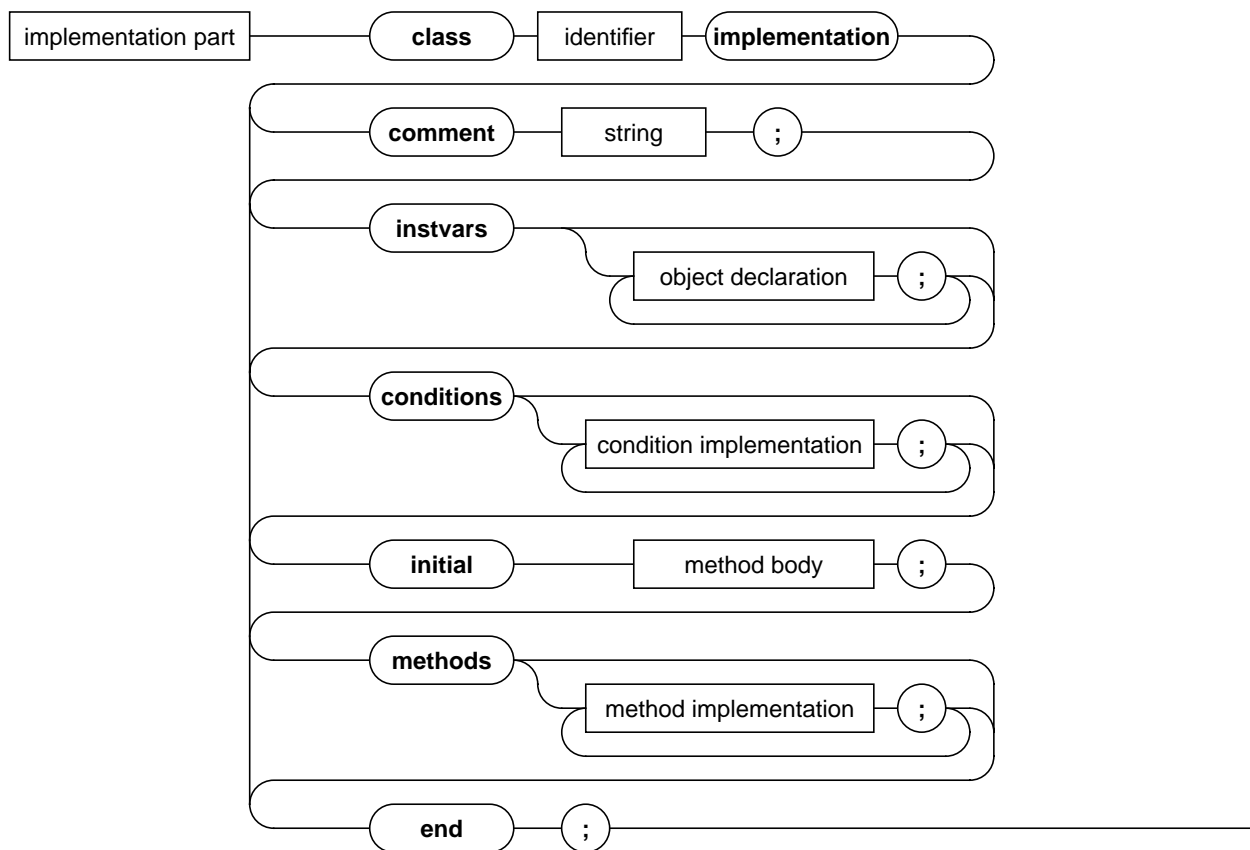
A.5 Main



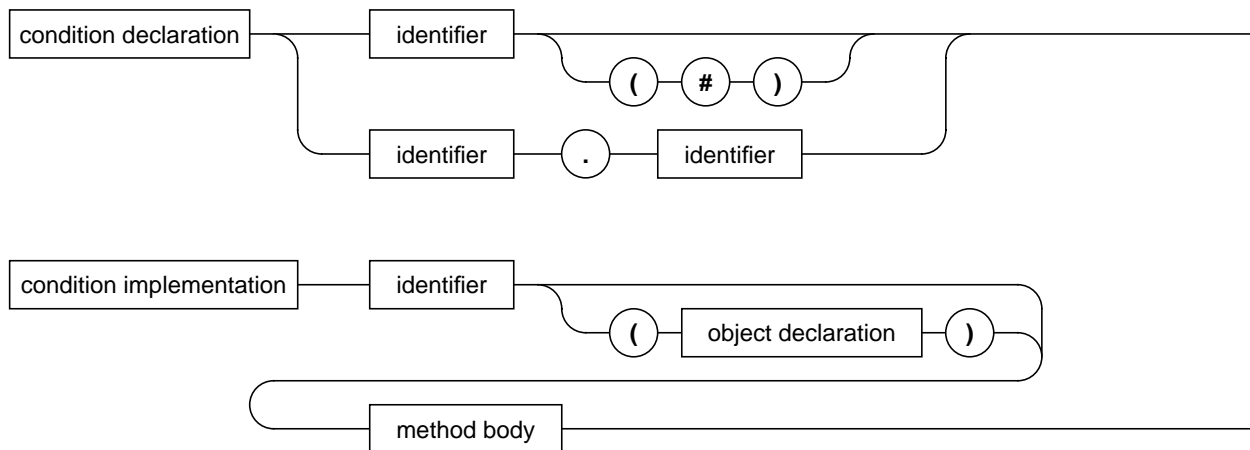
A.6 Class



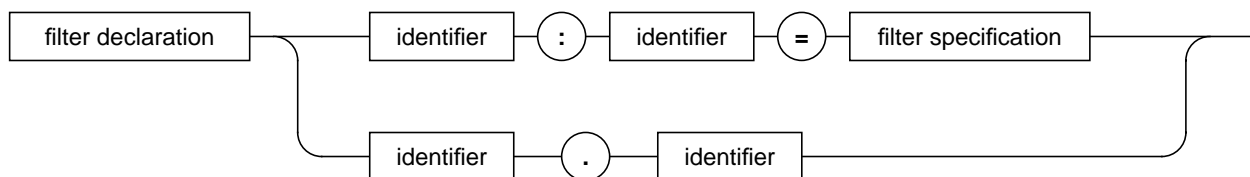


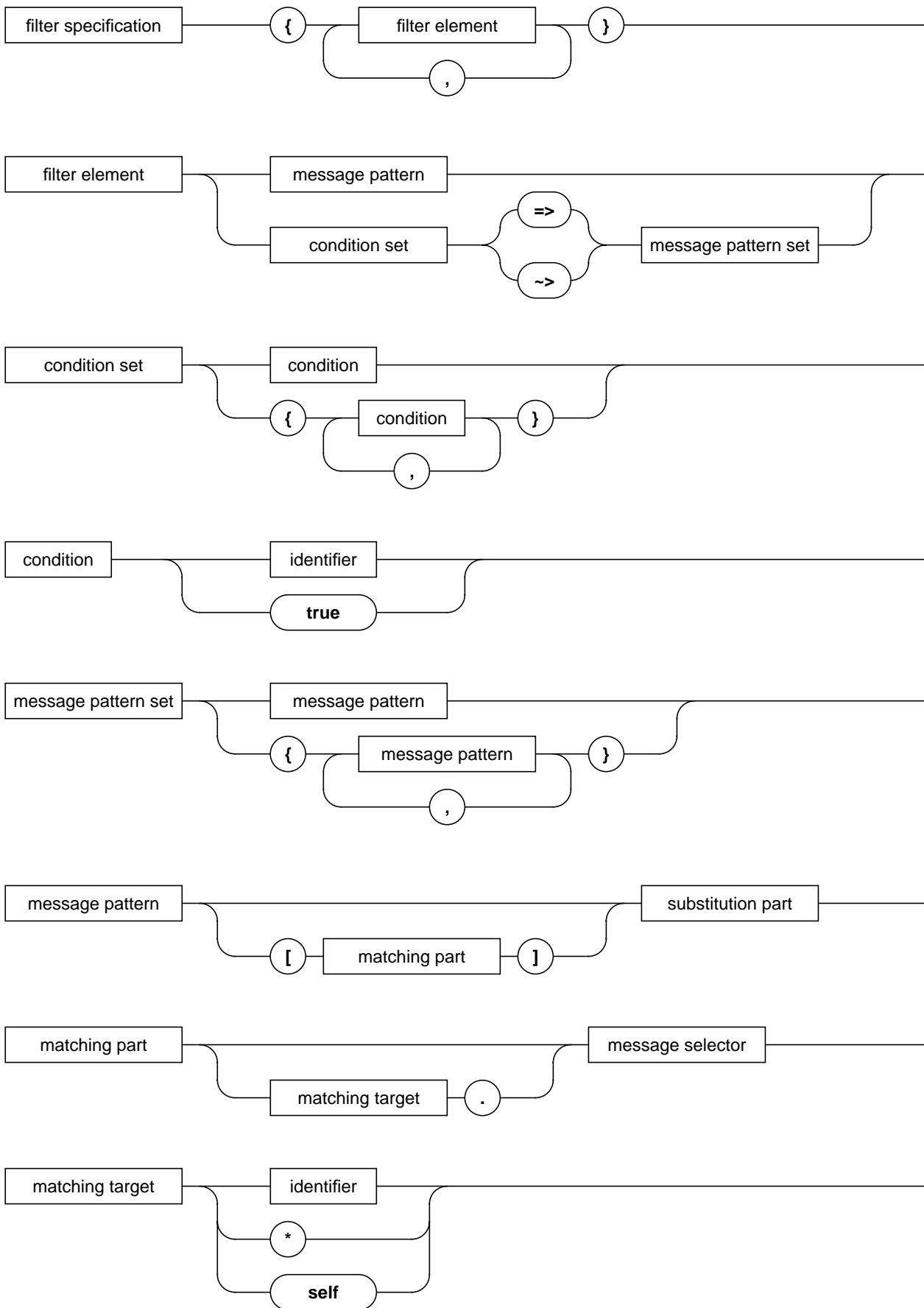


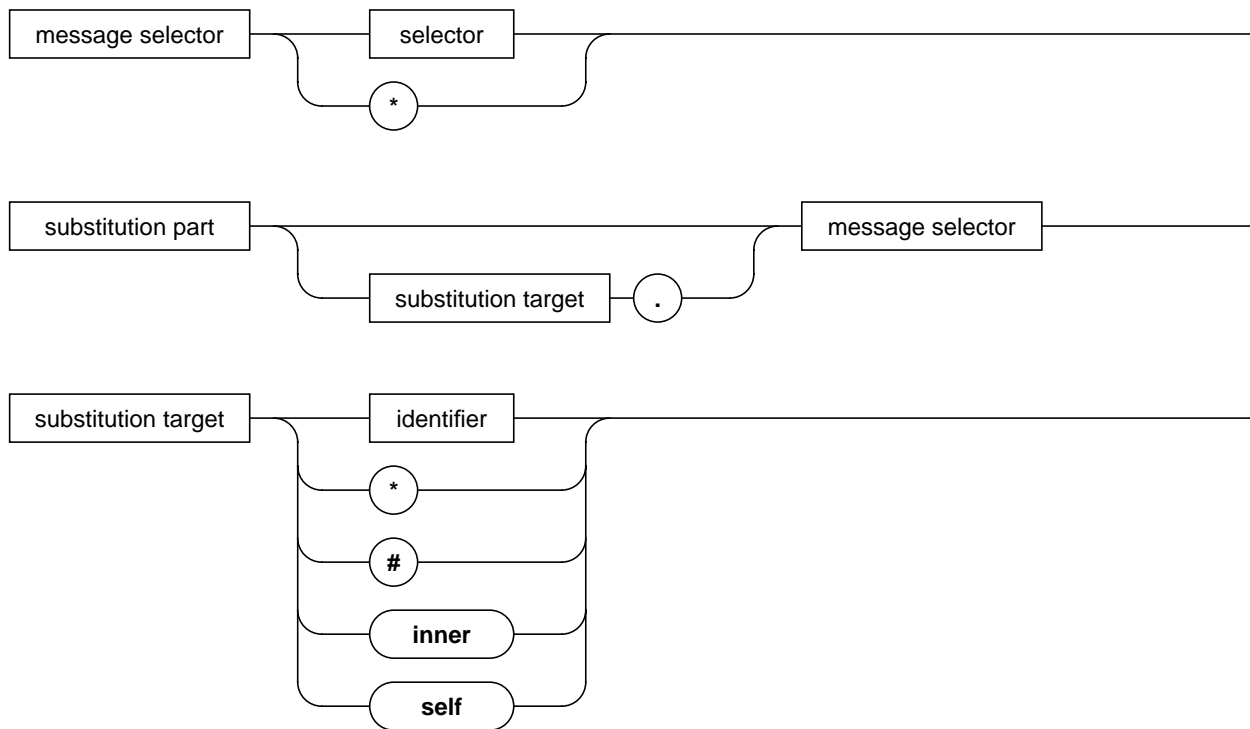
A.7 Conditions



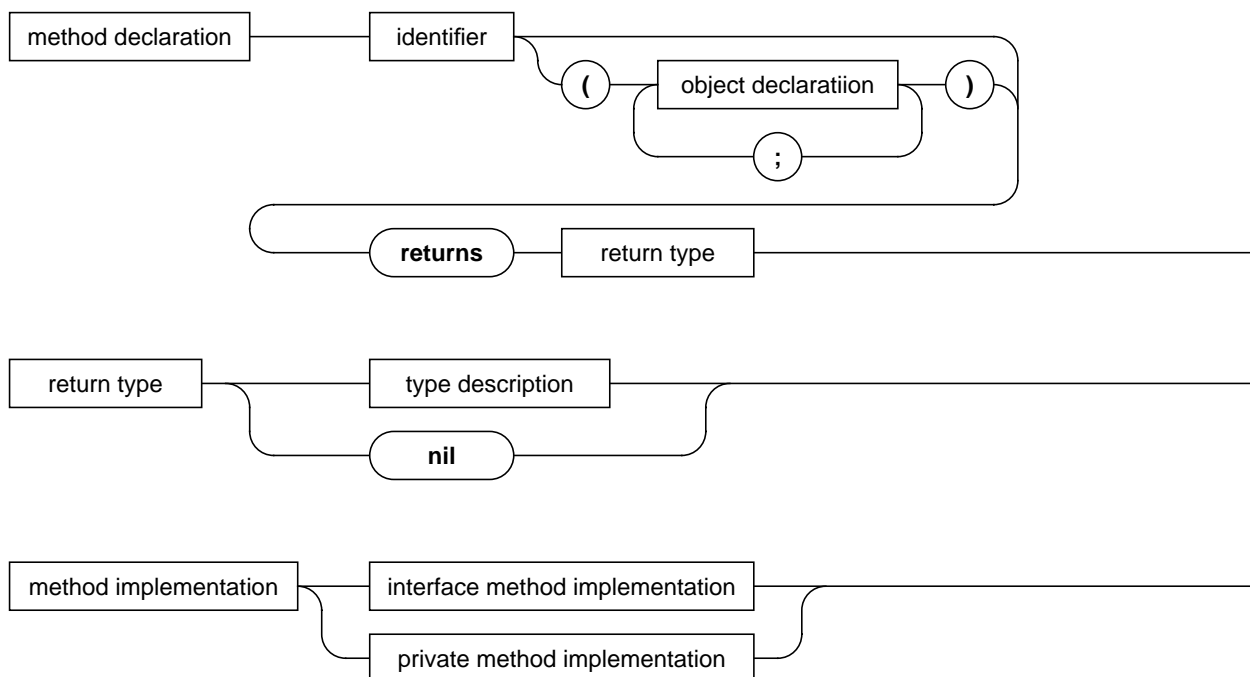
A.8 Filters

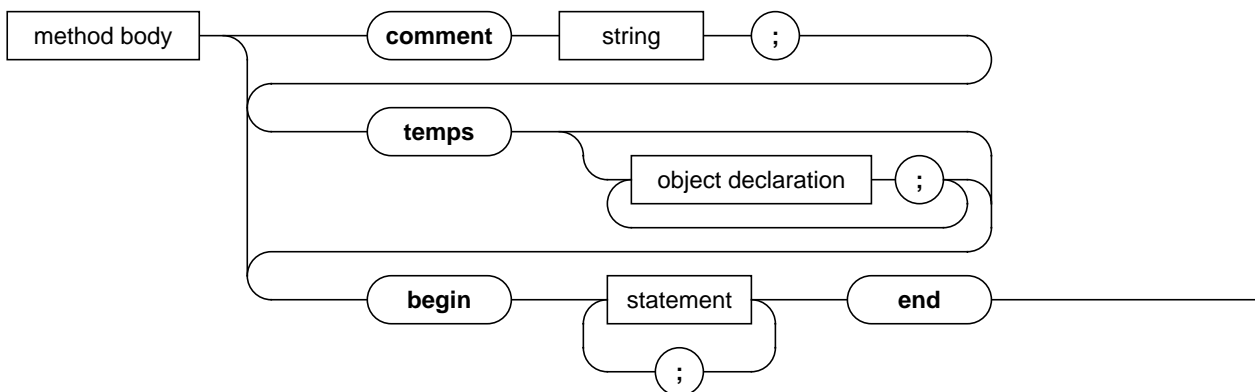
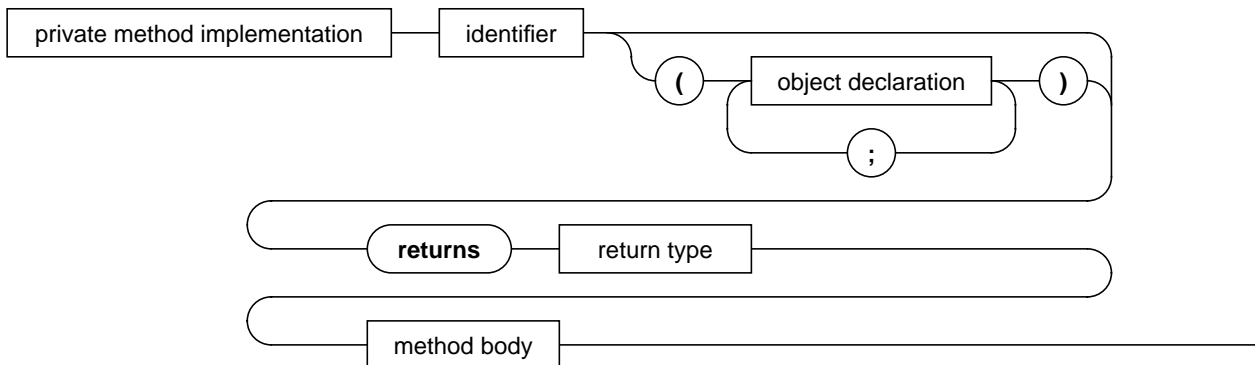
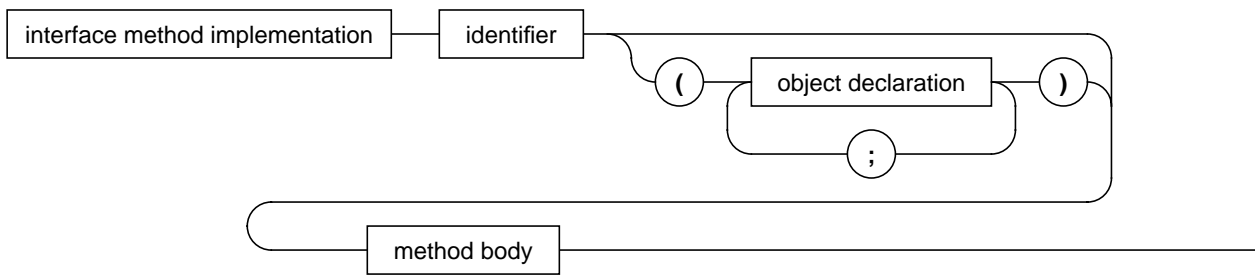




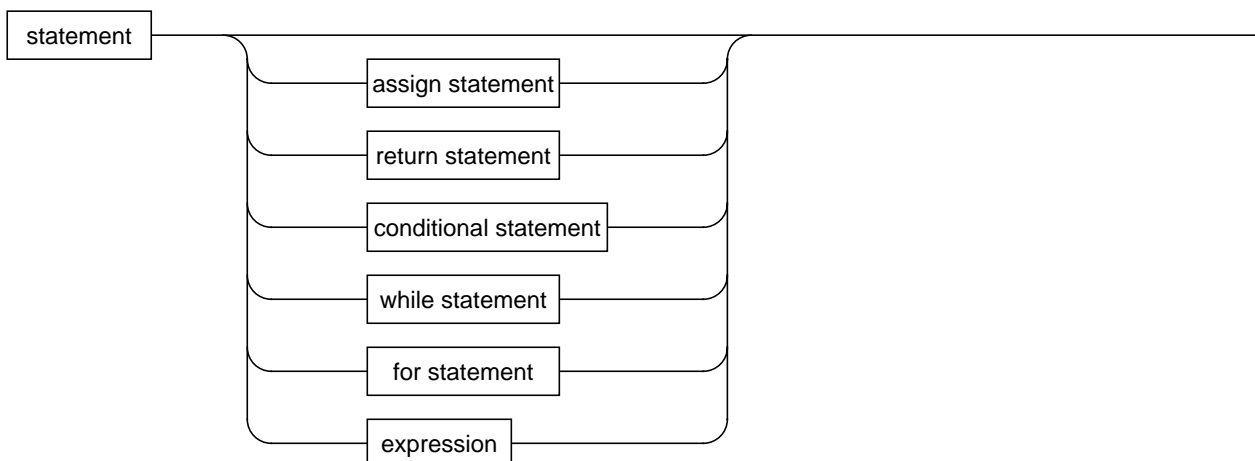


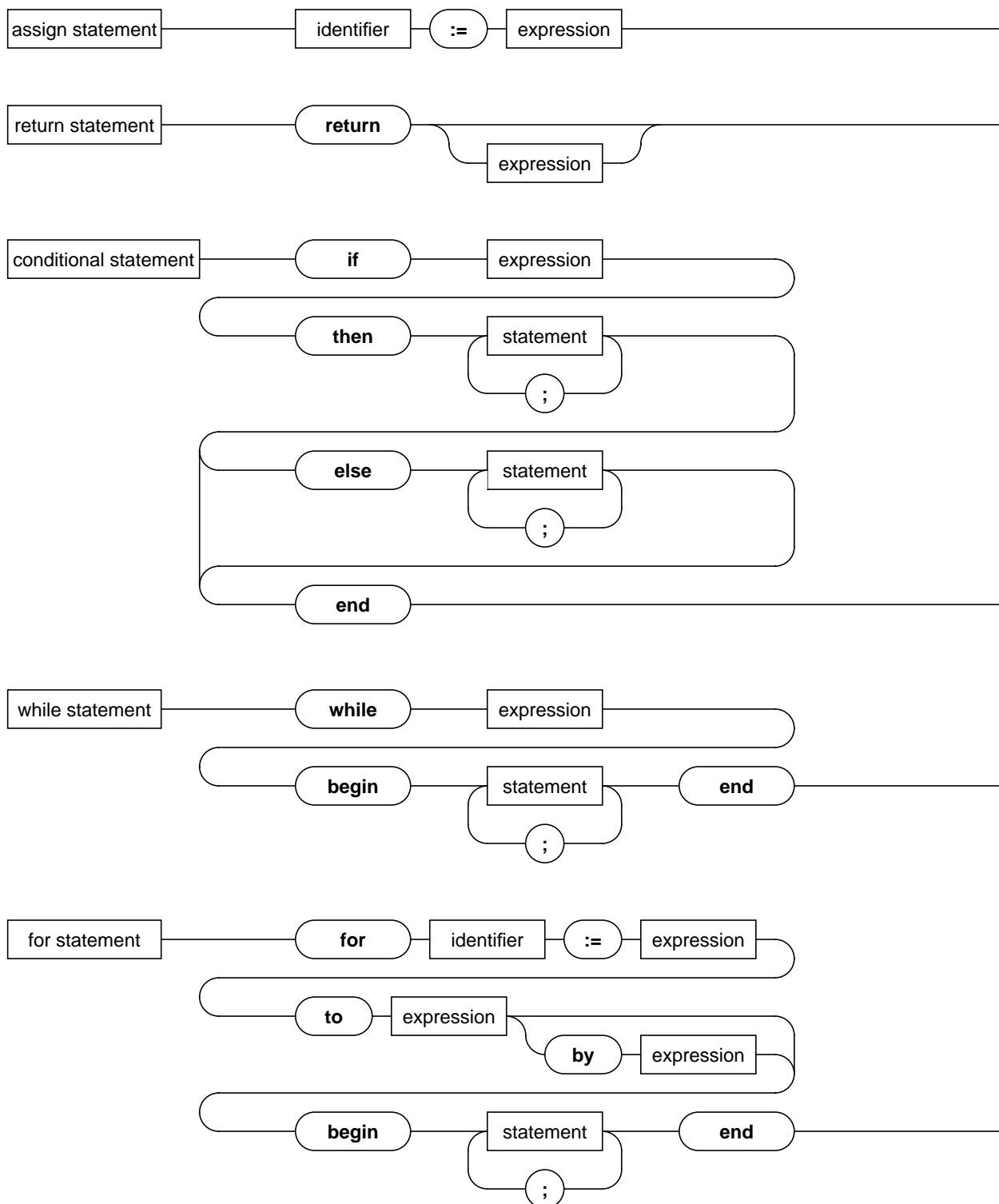
A.9 Methods



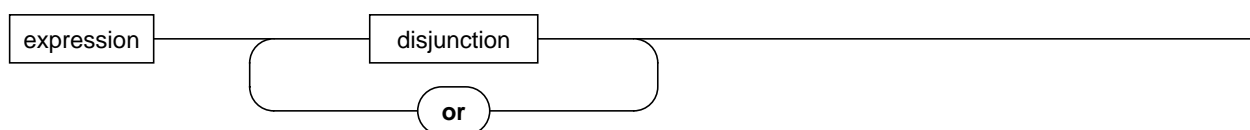


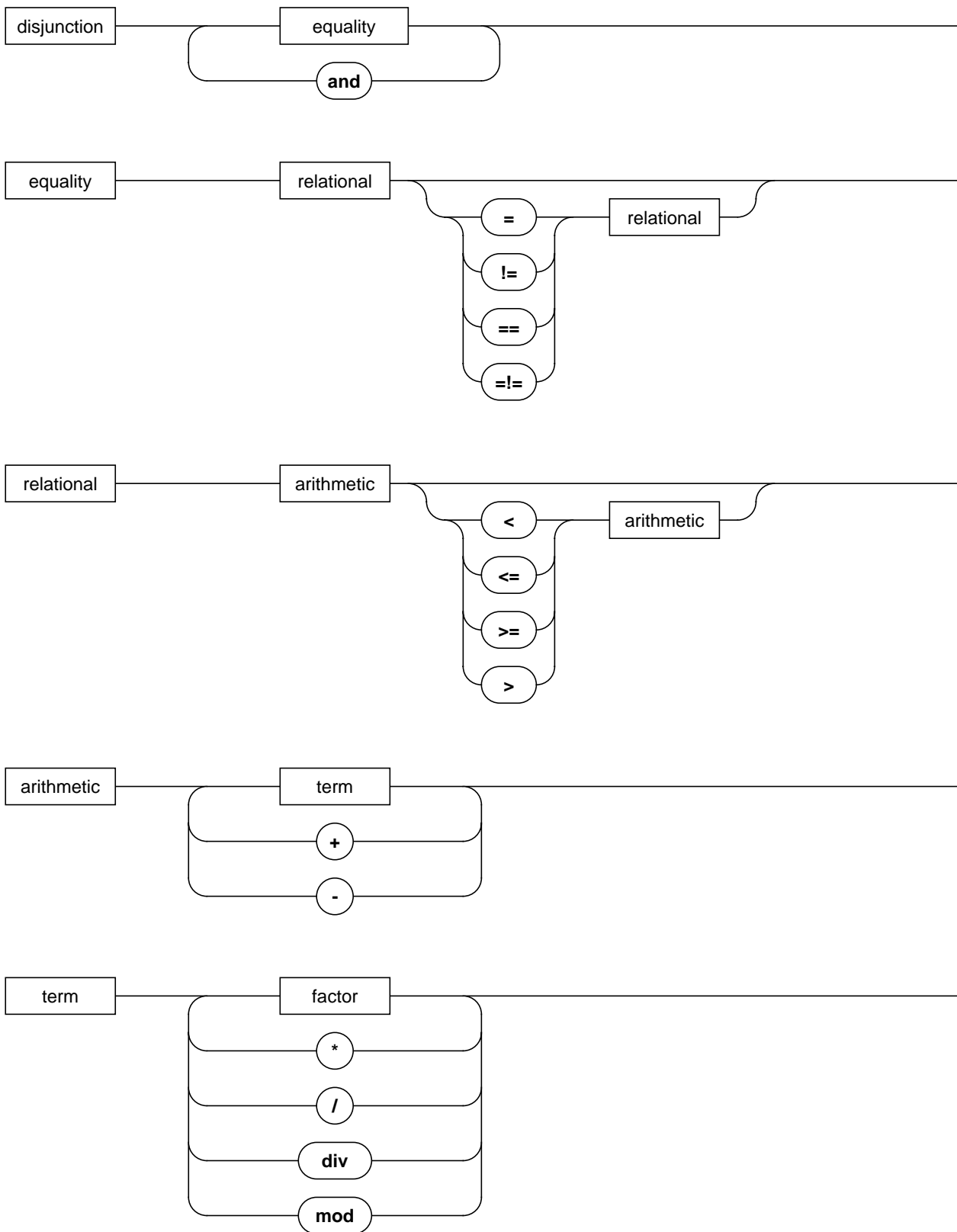
A.10 Statements

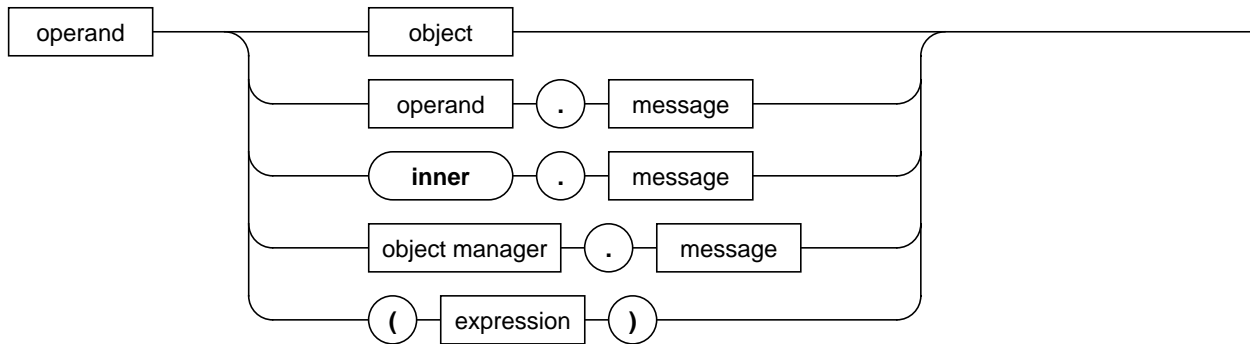
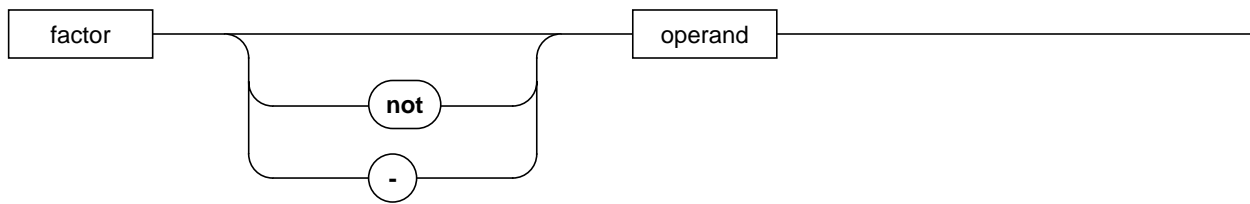




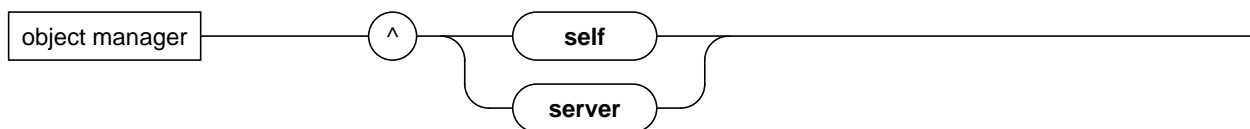
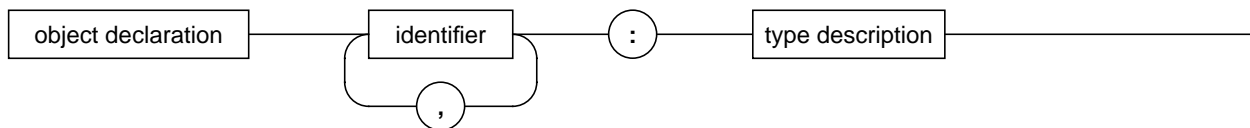
A.11 Expressions

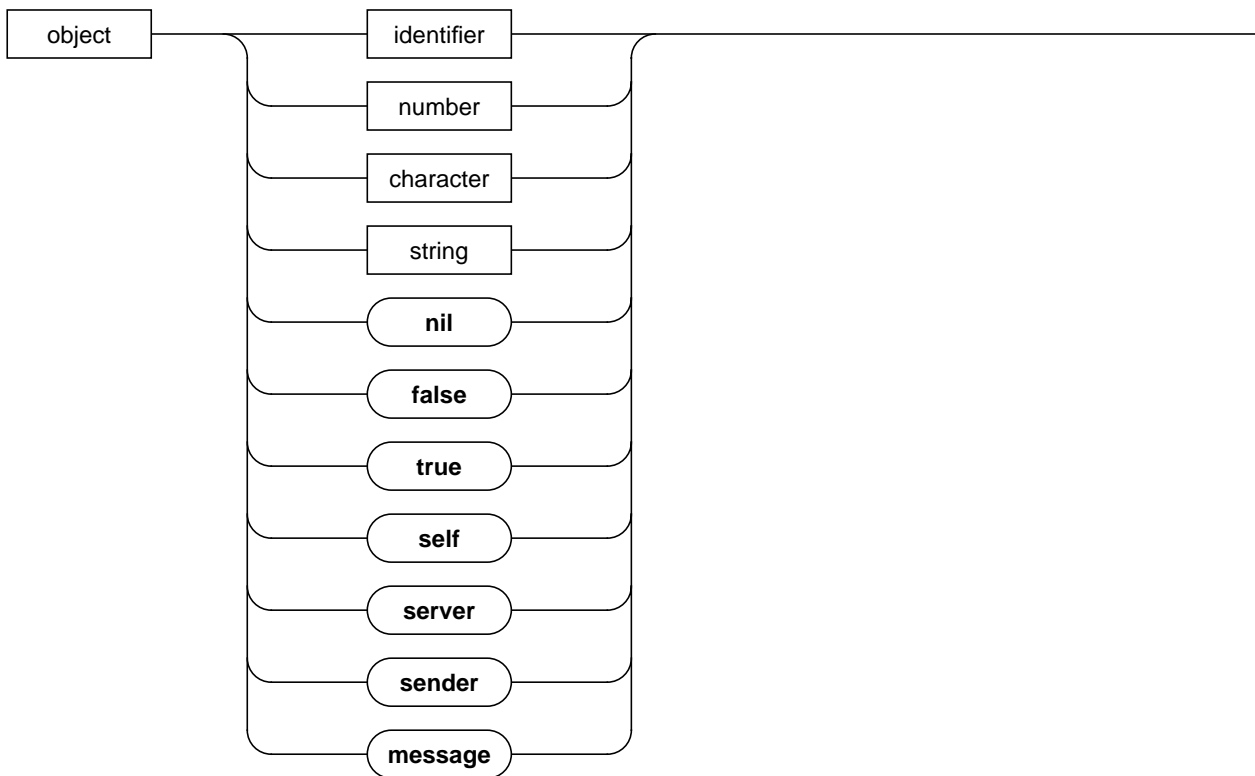




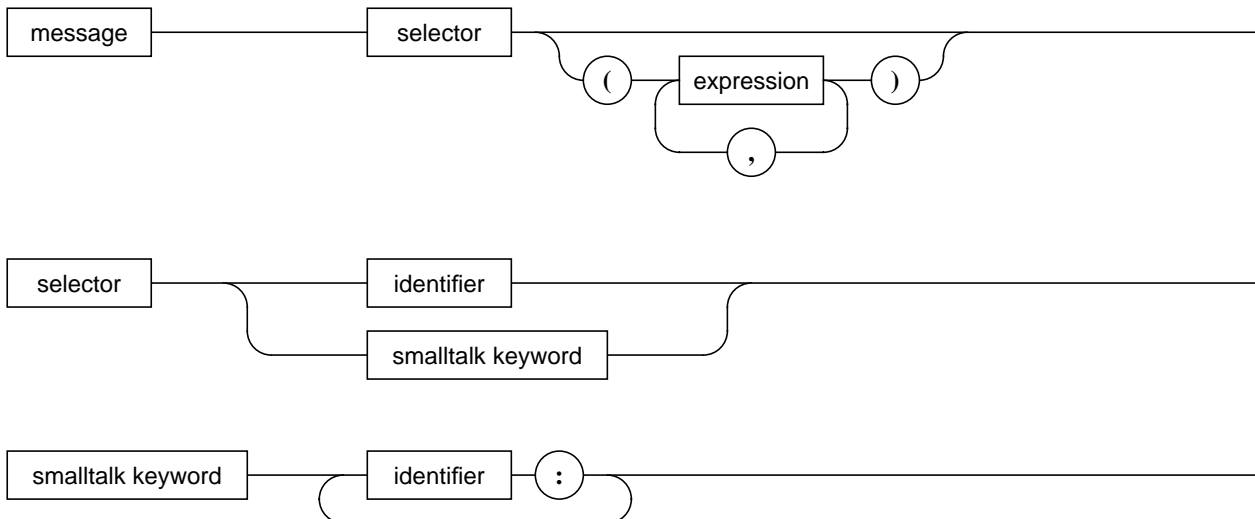


A.12 Objects





A.13 Messages



Sina Class Interfaces

B.1 SinaObject

```
#NoDefaultObject;
class SinaObject interface
comment
    'This class implements default behaviour.';
methods
//-----window access
    print(s : String) returns nil;
        // Print the argument on my window. When the window was not opened yet,
        // it will be opened.
    printLine(s: String) returns nil;
        // Print the argument on my window followed by a CR. When the window
        // was not opened yet, it will be opened.
    cr returns nil;
        // Print a newline character on my window. When the window was not
        // opened yet, it will be opened.
    tab returns nil;
        // Print a tab character on my window. When the window was not opened
        // yet, it will be opened.
    openWindow returns nil;
        // Open or reopen my window.
//-----error generation
    error(errorString : String) returns nil;
        // Generate an error.
        // The argument will be displayed on my window. Then, the thread that
        // invoked this method will be aborted.
//-----dialogs
    warn(warning : String) returns nil;
        // Display a warning message.
        // Opens a dialog window displaying the argument; this method finishes
        // when the user hits the 'ok' box or types cr in the window.
    request(prompt : String) returns String;
        // Request a String to be typed from the keyboard.
        // Opens a dialog window displaying the argument; the user then has to
        // enter a String until he types cr; this method then answers the String
        // entered by the user.
//-----string conversion
    toString returns String;
        // Answer a String whose characters are a description of the server.
//-----class membership
    class returns Class;
        // Answers the class of the server.
//-----inspecting
    inspect returns server;
        // Create an Inspector window on the server in which the user can examine
        // the server's variables.
//-----testing
```

```

isNil returns Boolean;
    // Tests if the server is the nil object: since this is not the case, this
    // method always returns false.
notNil returns Boolean;
    // Tests if the server is not the nil object: since this is not the case, this
    // method always returns true.
//----comparing
equal(anObject : Any) returns Boolean;
    // Answer whether the server and the argument represent the same
    // object, that is, have the same value.
unequal(anObject : Any) returns Boolean;
    // Answer whether the receiver and the argument do not represent
    // the same object.
    // This method implemented as
    //     return server.equal(anObject).not
same(anObject : Any) returns Boolean;
    // Answer true if the server and the argument, anObject, are the same
    // object (have the same object pointer) and false otherwise.
different(anObject : Any) returns Boolean;
    // Answer true if the server and the argument, anObject, are not the
    // same object (have the same object pointer) and false otherwise.
    // This method implemented as
    //     return server.same(anObject).not
inputfilters
    myBehavior : Dispatch = {inner.*}
end; // interface of class SinaObject

```

B.2 ActiveMessage

```

#NoDefaultObject;
class ActiveMessage interface
comment
    'This class represents an active message, that is, a message that is executing.
    Attributes of an active message include:
    - selector
    - arguments
    - sender
    - server
    - receiver';
methods
//----accessing attributes
sender returns Any;
    // Answers my sender object.
server returns Any;
    // Answers my server object .
receiver returns Any;
    // Answers my receiver object.
selector returns String;
    // Answers my selector.
numArgs returns SmallInteger;
    // Answers the number of my arguments.
args returns Array;
    // Answers an array containing my arguments.
argsAt(n : SmallInteger) returns Any;
    // Answers my n'th argument.
//----testing
isRecursive returns Boolean;
    // Answers whether this message is a recursive message, i.e. a message
    // sent to self, server or sender.
argsOfAtEqual(sel : String; i : SmallInteger; anObject : Any) returns Boolean;
    // Answer true if my selector is 'sel' and my i'th argument is equal to
    // 'anObject'. This method is implemented as:
    //     return self.selector = sel and self.argsAt(i) = anObject

```

```

isNil returns Boolean;
    // Tests if the server is the nil object: since this is not the case, this
    // method always returns false.
notNil returns Boolean;
    // Tests if the server is not the nil object: since this is not the case, this
    // method always returns true.
//----comparing
equal(anObject : Any) returns Boolean;
    // Answer whether the receiver and the argument represent the same
    // object.
unequal(anObject : Any) returns Boolean;
    // Answer whether the receiver and the argument do not represent
    // the same object.
    // This method implemented as
    //     return not server.equal(anObject)
same(anObject : Any) returns Boolean;
    // Answer true if the server and the argument, anObject, are the same
    // object (have the same object pointer) and false otherwise.
different(anObject : Any) returns Boolean;
    // Answer true if the server and the argument, anObject, are not the
    // same object (have the same object pointer) and false otherwise.
    // This method implemented as
    //     return not server.same(anObject)
//----class membership
class returns Class;
    // Answers the class of the server.
inputfilters
    myBehavior : Dispatch = {inner.*};
end; // interface of class ActiveMessage

```

B.3 SinaMessage

```

#NoDefaultObject;
class SinaMessage interface
comment
    'This class represents a reified message. It inherits the behavior from class
    ActiveMessage and adds behavior for changing the message's attributes
    and for dereification';
internals
    activeMessage : ActiveMessage;
methods
//----dereification
    continue returns nil;
        // Causes me to be (re)activated. Does not wait for my reply.
    fire returns nil;
        // Same as the method 'continue': causes me to be (re)activated.
        // Does not wait for my reply.
    send returns Any;
        // Causes me to be (re)activated. Waits until I have received a reply.
        // Answers my reply.
    sendContinue returns Any;
        // Causes me to be (re)activated. Waits until I have received a reply.
        // Then, I will be reactivated a second time. Answers my reply.
        // In fact, the message expression
        //     result := aMsg.sendContinue;
        // is equivalent to
        //     result := aMsg.send; aMsg.continue;
    reply(anObject : Any) returns nil;
        // Returns anObject as the reply to the sender of myself.
//----copying
    copy returns SinaMessage;
        // Answers a copy of myself, which has the same selector, server,
        // receiver and arguments as me, but who's sender is undefined (nil).

```

```
//-----changing message attributes
setSender(senderObject : Any) returns nil;
    // Changes my sender to be the argument, senderObject.
setServer(serverObject : Any) returns nil;
    // Changes my server to be the argument, serverObject.
setReceiver(receiverObject : Any) returns nil;
    // Changes my receiver to be the argument, receiverObject.
setSelector(selector : String) returns nil;
    // Changes my selector to be the argument, selector.
setArgs(args : Array) returns nil;
    // Changes my arguments to be the argument, args.
argsAtPut(n : SmallInteger; anObject : Any) returns nil;
    // My n'th argument will become the argument, anObject.
inputfilters
    myBehavior : Dispatch = {activeMessage.*, inner.*};
end; // interface of class SinaMessage
```

B.4 ObjectManager

```
#NoDefaultObject;
class ObjectManager interface
methods
    active returns SmallInteger;
        // Answers the number of active method invocations in the object .
    activeForMethod(selector : String) returns SmallInteger;
        // Answers the number of active invocations of the method given by
        // the argument .
    blockedReq returns SmallInteger;
        // Answers the number of messages that have been blocked by an
        // input wait filter.
    blockedInv returns SmallInteger;
        // Answers the number of messages that have been blocked by an
        // output wait filter
inputfilters
    myBehavior : Dispatch = {inner.*};
end; // interface of class ObjectManager
```

APPENDIX

C

Using Smalltalk
classes in *Sina/st***C.1 *Sina/st* variable declaration using a Smalltalk class**

A variable *v* that is declared with the *Sina/st* variable declaration *v* : C; will have an initial value (an object) assigned to it. This initial value will be an instance of the class C. Table C-1 lists the initial values for some Smalltalk classes. When the class C is not listed in this table and C is a Smalltalk class, it will be initialized according to table C-2.

TABLE C-1

Initial value of the variable v in the variable declaration v : C; ^a.

Class C	Initial value of v	Class of v
Number	0	SmallInteger
Fraction	0/1	Fraction
Integer	0	LargePositiveInteger
LargeNegativeInteger	0	LargePositiveInteger
LargePositiveInteger	0	LargePositiveInteger
SmallInteger	0	SmallInteger
LimitedPrecisionReal	0	SmallInteger
Double	0.0d	Double
Float	0.0	Float
Point	0@0	Point
Character	Character value: 0	Character
Boolean	false	False
True	true	True
False	false	False
UndefinedObject	nil	UndefinedObject

a. Note, that sometimes the class of *v* is not C. This is because instances of subclasses of ArithmeticValue will be created and initialized with the Smalltalk message zero. These subclasses include Number, LimitedPrecisionReal, Double, Float, Fraction, Integer, SmallInteger, LargeNegativeInteger, LargePositiveInteger and Point.

TABLE C-2

Sina variable declaration using a Smalltalk class C

Sina variable declaration	Smalltalk equivalent
<i>v</i> : C;	<i>v</i> := C new.
<i>v</i> : C(<i>Expr</i>);	<i>v</i> := C new: <i>Expr</i> .

TABLE C-2

Sina variable declaration using a Smalltalk class C

Sina variable declaration	Smalltalk equivalent
<code>v : C(Expr1, Expr2, ..., ExprN);</code>	<code>v := C new: Expr1 with: Expr2 ... with: ExprN.</code>
<code>v : C keyw;</code>	<code>v := C keyw.</code>
<code>v : C keyw1:...keywN:(Expr1, ..., ExprN);</code>	<code>v := C keyw1: Expr1 ... keywN: ExprN.</code>

Examples:

```
Sina
n      : SmallInteger;
point1 : Point;
point2 : Point x:y:(260,435);
arr1    : Array;                      // size not specified
arr2    : Array(n+2);                 // initial size given
arr3    : Array with:with:with:(3, ' element', 'array'); //initialized array
arr4    : TypedArray(6,SmallInteger); // this class has not been defined
today   : Date today;
circle  : Circle center:radius:(point2, 50);

Smalltalk
n      := 0;
point1 := 0@0;
point2 := Point x: 260 y: 435.
arr1   := Array new.
arr2   := Array new: n+2.
arr3   := Array with: 3 with: 'element' with: 'array'.
arr4   := TypedArray new: 6 with: SmallInteger.
today  := Date today.
circle := Circle center: point2 radius: 50.
```

C.2 Smalltalk selectors

Smalltalk provides three types of selectors

- unary, consisting of alphanumeric characters only, eg. `today`;
- binary, consisting of one or two non-alphanumeric characters, eg. `+`;
- and keyword, consisting of a sequence of alphanumeric characters plus a colon, eg. `center:radius:.`

In *Sina/st*, a selector is allowed to consist of alphanumeric or colon characters only. Therefore, Smalltalk unary and keyword selectors can be used in *Sina/st* unchanged, but binary selectors must be converted to an equivalent selector as listed in the center column of table C-3. Alternatively, you can use an operator expression with an operator shown in the right column.

TABLE C-3

Smalltalk binary selectors and their equivalents in Sina/st

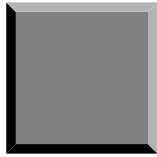
For the Smalltalk selector	Use the Sina selector	Or use an operator expression with the Sina operator
<i>arithmetic</i>		
+	plus	+
-	minus	-
*	times	*
/	divide	/
//	div	div
\\	mod	mod
negated	negated	-
<i>logical</i>		
&	and	and
	or	or
not	not	not
<i>comparing</i>		
>	greater	>
>=	atleast	>=
<	less	<
<=	upto	<=
<i>testing</i>		
=	equal	=
~=	unequal	!=
==	same	==
~~	different	!=
<i>miscellaneous</i>		
,	cat	
printString	toString	
@	pointWith	
->	associatesWith	



References

- [Aksit88] Mehmet Aksit and Anand Tripathi. “Data abstraction mechanisms in sina/st.” In Meyrowitz, editor, *OOPSLA '88, Conference Proceedings*, volume 23 of *SIGPLAN Notices*, pages 267–275, November 1988. ISBN 0-89791-284-5, ISSN 0362-1340.
- [Aksit89] Mehmet Aksit. *On the Design of the Object-Oriented Language Sina*. Ph.D. dissertation, University of Twente, Dept. of Computer Science, P.O. box 217, 7500 AE, Enschede, The Netherlands, 1989. ISBN 90-365-0250-0.
- [Aksit91] Mehmet Aksit, Jan-Willem Dijkstra, and Anand Tripathi. “Atomic Delegations: Object-Oriented Transactions.” *IEEE Software*, pages 84–92, March 1991. IEEE Computer Society, 10622 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1264, USA. ISSN 0740-7459.
- [Aksit92a] Mehmet Aksit and Lodewijk Bergmans. Obstacles in object-oriented software development. In Andreas Paepcke, editor, *OOPSLA '92, Conference Proceedings*, volume 27 of *ACM SIGPLAN Notices*, pages 341–358, October 1992. ISSN 0362-1340, ISBN 0-201-53372-3.
- [Aksit92b] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming*, number 615 in *Lecture Notes in Computer Science*, pages 372–395, Berlin Heidelberg, June/July 1992. European Conference on Object-Oriented Programming, Springer-Verlag. ISBN 0-387-55668-0 / ISBN 3-540-55668-0.
- [Aksit93] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *ECOOP '93, Workshop on Object-Based Distributed Programming*, number 791 in *Lecture Notes in Computer Science*, pages 152–184, Berlin Heidelberg, July 1993. European Conference on Object-Oriented Programming, Springer-Verlag. ISBN 0-387-57932-X / ISBN 3-540-57932-X.
- [Aksit94] Mehmet Aksit, Jan Bosch, William van der Sterren, and Lodewijk Bergmans. Real-time specification inheritance anomalies and real-time filters. In Mario Tokoro and Remo Pareschi, editors, *ECOOP '94, Object-Oriented Programming*, number 821 in *Lecture Notes in Computer Science*, pages 386–407, Berlin Heidelberg, July 1994. European Conference on Object-Oriented Programming, Springer-Verlag. ISBN 0-387-58202-9 / ISBN 3-540-58202-9.
- [Aksit95] Mehmet Aksit and Lodewijk Bergmans. “Application Domains and Obstacles in Conventional Object-Oriented Software Development.” Draft version. University of Twente, Dept. of Computer Science, P.O. box 217, 7500 AE, Enschede, The Netherlands, 1995.

- [Algra94] Egbert Algra. "Process programming and learning domains in object oriented hermeneutic software development." M.Sc. thesis, University of Twente, Dept. of Computer Science, P.O. box 217, 7500 AE, Enschede, The Netherlands, June 1994.
- [Bergmans94a] Lodewijk Bergmans. *Composing Concurrent Objects*. Ph.D. dissertation, University of Twente, Dept. of Computer Science, P.O. box 217, 7500 AE, Enschede, The Netherlands, June 1994. ISBN 90-9007359-0.
- [Bergmans94b] Lodewijk M.J. Bergmans. The Composition-Filters Object Model. This paper was derived from Chapter 2 of [Bergmans94a]. University of Twente, PO box 217, 7500 AE Enschede, The Netherlands, June 1994.
- [vDijk95] Wietze van Dijk and John Mordhorst. "CFIST: Composition Filters in Smalltalk." HIO Graduation thesis, July 1995.
- [Ellis90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1990. ISBN 0-201-51459-1.
- [Goldberg83] Adele Goldberg and David Robson. *Smalltalk-80: The Language And Its Implementation*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, USA, 1983. ISBN 0-201-11371-6.
- [Goldberg89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, MA, USA, September 1989. ISBN 0-201-13688-0.
- [Marcelloni95] Francesco Marcelloni and Mehmet Aksit. "Design and Implementation of the Object-Oriented Fuzzy-Logic Reasoning Framework." Draft version, University of Twente, P.O.Box 217, 7500 AE Enschede, The Netherlands, 1995.
- [ParcPlace92a] ParcPlace Systems, 999 E. Arques Ave., Sunnyvale, CA 94086-4593, USA. *Objectworks\Smalltalk Release 4.1 User's Guide*, 1992.
- [ParcPlace92b] ParcPlace Systems, 999 E. Arques Ave., Sunnyvale, CA 94086-4593, USA. *VisualWorks Release 1.0 User's Guide*, 1992.
- [Tekinerdogan94] Bedir Tekinerdogan. "The design of an object-oriented framework for atomic transactions." M.Sc. thesis, University of Twente, Dept. of Computer Science, P.O. box 217, 7500 AE, Enschede, The Netherlands, March 1994.
- [Ungar87] David Ungar and Randall B. Smith. "Self: The Power of Simplicity." In *OOPSLA '87, Conference Proceedings*, pages 227-242.
- [Vuijst94] Charles Vuijst. "Design of an Object-Oriented Framework for Image Algebra." M.Sc. thesis, University of Twente, Dept. of Computer Science, P.O. box 217, 7500 AE, Enschede, The Netherlands, December 1994.
- [Wegner87] Peter Wegner. Dimensions of object-based language design. In Norman Meyrowitz, editor, *OOPSLA '87 Conference Proceedings*, volume 22 of *SIGPLAN Notices*, pages 168-182, December 1987. ISBN 0-89791-247-0.



Index

Symbols

#Category 49
 #DefaultObject 49
 #DefBehaviorFilter 50
 #DefSyncFilter 49
 #NoDefaultObject 49
 #NoDefBehaviorFilter 50
 #NoDefSyncFilter 49
 <operate> button 13
 <operate> menu 13
 <select> button 13
 <window> button 13
 ^self (pseudo variable) 38
 messages to ~ 70
 see also object manager
 ^server (pseudo variable) 38
 messages to ~ 70
 see also object manager

A

abstract communication type 81
 ACT 81
 active message 82, 85
 ActiveMessage (class) 83, 114
 Any (type) 34
 assignment statement 36

B

behavior injection 5

C

character literal 32
 class 47, 102
 ~ parameter 33
 ~ switch 48
 declaration 47
 implementation part 48
 instance of a ~ 51
 interface part 47
 comment 31
 composition filters object model 4, 23
 concurrency 53, 84
 condition 24, 65, 104
 ~ in filterinitializer 72

conditional statement 45
 continue (dereification method) 86, 87
 control flow 58
 requirements 59
 ~ for the conditional statement 60
 ~ for the while and for statement 60
 control structure 45
 conditional statement 45
 for statement 45
 while statement 46

D

dereification 85, 86
 dispatch filter 5, 77, 78

E

early return 59, 63
 error filter 5, 77, 78
 exclusionelement 72
 expression 39, 108
 message expression 40
 operator expression 41
 external 24, 33

F

false 33
 filter 104
 ~condition 72
 ~element 71
 exclusionelement 72
 inclusionelement 72
 rewrite rules 73
 ~handler 77
 dispatch 77, 78
 error 77, 78
 meta 77, 79
 send 77, 78
 wait 77, 78
 ~initializer 71
 declaration 71
 dispatch ~ 5, 77, 78
 error ~ 5, 77, 78
 input filters 24, 70
 local ~ 71
 messagepattern 72

- matchingpattern 72
 - signaturepattern 72
- meta ~ 5, 77, 79
- output filters 24, 70
- reused ~ 71
- send ~ 5, 77, 78
- wait ~ 5, 77, 78
- first-class object 30
- floating point literal 32
- for statement 45

G

- global external 36

I

- implementation part 27
- inclusion element 72
- initial method 56
- inner (pseudo variable) 38
 - during method invocation 63
 - messages to ~ 70
- input filters 24, 70
- instance variable 24, 33
- integer literal 32
- interface method 53
- interface part 26
- internal 24, 33

L

- literal 32, 100
 - character 32
 - floating point 32
 - integer 32
 - number 32
 - string 32
- local variable 33

M

- main 102
- main method 57
- matchingpattern 72
- message 111
 - active ~ 82, 85
 - dereification 85, 86
 - continue 86, 87
 - reply 87, 88
 - send 87, 90
 - filtering a ~ 74
 - recursive ~ 83
 - reification 81, 85, 85
 - reified ~ 85
 - sending a ~ 81
- message (pseudo variable) 38, 83
 - during condition invocation 67
 - during method invocation 63
 - messages to ~ 70
- message expression 40

- messagepattern 72
 - matchingpattern 72
 - signaturepattern 72
- meta filter 5, 77, 79
- method 24, 106
 - ~ invocation 62
 - ~ parameter 33
 - control flow in a ~ 58
 - early return ~ 59
 - invocation of an ~ 63
 - initial ~ 56
 - interface ~ 53
 - main ~ 57
 - normal ~ 59
 - invocation of a ~ 63
 - private ~ 56
 - returning a value from a ~ 57
 - returntype 53

N

- nil 33
- normal return 59, 63

O

- object manager 38, 52
 - ^self 38
 - ^server 38
- ObjectManager (class) 52, 116
- operator expression 41
- output filters 24, 70
- owner 30

P

- private method 56
- pseudo variable 37
 - ^self 38
 - ^server 38
 - inner 38
 - message 38, 83
 - messages to ~ 70
 - self 38
 - sender 38
 - server 38

R

- reification 85, 85
- reified message 85
- reply
 - (dereification method) 87, 88
 - (method return) 58
- return
 - early ~ 59, 63
 - normal ~ 59, 63
- return statement 58
- returntype 53

S

- self (pseudo variable) 38
 - during method invocation 63
 - messages to ~ 70
- send (dereification method) 87, 90
- send filter 5, 77, 78
- sender (pseudo variable) 38
 - during method invocation 63
 - messages to ~ 70
- server (pseudo variable) 38
 - during method invocation 63
 - messages to ~ 70
- signature of an object 76
- signaturepattern 72
- Sina (global external) 12, 12, 36, 37, 57
- SinaCompilerInterface 12
- SinaMessage (class) 94, 115
- SinaObject (class) 49, 113
- Smalltalk (system dictionary) 12
- statement 64, 107
 - assignment ~ 36
 - conditional ~ 45
 - for ~ 45
 - message expression 40
 - operator expression 41
 - return ~ 58
 - while ~ 46
- string literal 32

T

- thread 82, 83
- TRESE project 1
- true 33
- type
 - ~description 34
 - Any 34
 - method return~ 53

V

- variable 33
 - assignment 36
 - class parameter 33
 - declaration 34
 - external 33
 - global external 36
 - initial value 36
 - instance ~ 33
 - internal 33
 - local ~ 33
 - method parameter 33
 - typedescription 34

W

- wait filter 5, 77, 78
- while statement 46

